UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

BÚSQUEDA POR SIMILITUD EN ESPACIOS MÉTRICOS SOBRE PLATAFORMAS PARALELAS MULTI-CORE Y MULTI-GPU

SIMILARITY SEARCH IN METRIC SPACES ON PARALLEL MULTI-CORE AND MULTI-GPU PLATFORMS

TESIS DOCTORAL DE:
RICARDO JAVIER BARRIENTOS ROJEL

DIRIGIDA POR:
JOSÉ IGNACIO GÓMEZ PÉREZ
MANUEL PRIETO MATÍAS

Madrid, 2014

Búsqueda por Similitud en Espacios Métricos sobre Plataformas Paralelas multi-core y multi-GPU

Similarity Search in Metric Spaces on Parallel multi-core and multi-GPU Platforms

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

TESIS DOCTORAL

Ricardo Javier Barrientos Rojel

Madrid, Octubre de 2013

Búsqueda por Similitud en Espacios Métricos sobre Plataformas Paralelas multi-core y multi-GPU

Memoria presentada por Ricardo Javier Barrientos Rojel para optar al grado de Doctor por la Universidad Complutense de Madrid.

Similarity Search in Metric Spaces on Parallel multi-core and multi-GPU Platforms

Disertation submitted by Ricardo Javier Barrientos Rojel to the Complutense University of Madrid in partial fulfilment of the requirements for the degree of doctor of philosophy.

Directores/Advisors: José Ignacio Gómez Pérez
Manuel Prieto Matías

Madrid, 2 de Octubre del 2013.

A Telmo e Iris

A Ana

A Raysa y a mis futuros hijos...

Agradecimientos

No puedo contener mi emoción al comenzar a escribir estas líneas. Han sido varios los años que he dedicado para terminar la presente tesis, y en este tiempo he conocido gente maravillosa que me ha ayudado directa e indirectamente en este proceso.

En primer lugar, quiero agradecerles a mis padres, Telmo e Iris, de quienes he recibido un apoyo incondicional toda mi vida. Les agradezco su confianza en mí, por no dudar en mis capacidades. Por apoyarme desde el primer momento en realizar el Doctorado, aún sabiendo que esto me llevaría tan lejos de casa.

A Raysa, por estar conmigo día a día a lo largo de todo este tiempo, por tu constante fuerza y ánimo, sobre todo en aquellos momentos donde se combinaron gran cantidad de trabajo y soledad. Me siento realmente afortunado de haber encontrado a alguien como tú.

A Nacho, por haber tomado la decisión de dirigir mi tesis doctoral, y estar permanentemente pendiente de mi formación como Doctor. He tenido gran fortuna de haber sido dirigido por alguien tan dedicado a su trabajo, y al trabajo de sus alumnos, como lo eres tú. A Manuel y a Christian, por su apoyo desde mi comienzo en la UCM, y brindar permanente ayuda a todo lo relacionado con mi trabajo y formación profesional. A P. Zezula, por su siempre buena disposición y haberme acogido en una estancia de investigación.

A Roberto Uribe-Paredes y familia, por su apoyo y consejo constante. Por abrirme las puertas de su casa y haberme tratado como familia. Estaré siempre agradecido.

A Mauricio Marin, por siempre preocuparse de mi formación y desarrollo profesional.

A Mauricio y Mariana, dueños del maravilloso bar Chileno en Madrid: “El Regreso del Winnipeg”. Por recibirme siempre con los brazos abiertos y una sonrisa sincera. Gracias por su amistad mis amigos.

A todos mis amigos y compañeros de laboratorio, pues han hecho mucho más grata mi estadía.

Quiero agradecer formalmente al profesor Francisco Tirado y al Gobierno de España por haberme otorgado una beca FPI (TIN2008-00508) para realizar mis estudios de Doctorado.

Finalmente, quiero realizar un agradecimiento especial, a una persona que ya no se encuentra con nosotros, a mi abuela Ana. Gracias por fabricar los cimientos de nuestra familia. Gracias por haber sido partícipe de los más hermosos recuerdos de mi infancia. Te digo, hoy y siempre, Gracias.

Ricardo J. Barrientos, Octubre 2013.

Contents

Chapter 1

Introduction

In traditional databases, we usually search for results that make an exact match with the query, i.e. given a query, the structured data which are exactly equal to the query are retrieved. Moreover, with the evolution of the information and communication technologies, information repositories that cannot be structured in the traditional way have emerged. Data types, such as audio, video or images cannot be structured in key or records, but nowadays it is required the search on them. Therefore, new models are necessary to search on unstructured repositories, where the traditional *exact match* searching cannot be applied.

Looking for a solution, the first concept to take account is the *similarity search* [31], i.e. search of the elements of the database that are similar or close to the query. The similarity is measured with a *distance function* that must satisfies the properties of symmetry, positivity and triangle inequality, thus this latter function and the set of data is called *metric space*.

A very studied technique the last years to search on complex objects has been the *indexing* ([17, 60]) on metric spaces. Many indexes have been proposed for

sequential computation, which reach good efficiency in multidimensional spaces with a large number of elements. However, the design of these indexing techniques have been focused on the search of individuals queries in sequential computation, solving two main kind of queries, *range* and kNN queries. A range query with radius r and query q, represented as (q,r), is the operation that obtains from the database the set of objects whose distance to the query object q is not larger than the radius r. A kNN query (namely k nearest-neighbors query), represented as kNN(q) retrieves as result the k nearest elements (of the database) from q.

Since the problem of similarity search has arisen in many different fields, a plethora of different solutions have been proposed from unrelated areas such as statistics, computational geometry, artificial intelligence, databases, computational biology, pattern recognition, data mining, the Web. Nowadays the Web engines index many billions of documents and hundred of millions of other complex objects such as multimedia data. The workloads in the large search engines are distinguished by a large quantity of queries being processed all the time on a large quantity of data (hundreds of millions). In this kind of search engine the metric to optimize is the throughput, which is defined as the number of completely solved queries per time unit. Recently has appeared the first commercial search engine (*Google Goggles* [2]), which allows to input an image as query, and despite that just work well with a certain kind of objects, is the beginning of this kind of applications.

To reach high throughput and response rates on hundreds of million of objects with thousands of queries per second, it is necessary to use parallel computing. Current implementations that use parallel computing, are performed on hundreds of processors (or *nodes*), where the objects and indexes are distributed, and where each node could be composed by a set of CPU-cores and GPUs. The main contribution

of the present thesis is focused on the efficient search in metric spaces on one of the nodes aforementioned, using an environment of shared memory.

We used as basis metric indexes that already existed in the technique literature, which have implemented efficient sequential algorithms to process queries in metric spaces. As mentioned above, the metric indexes have been optimized to solve individual queries, and not to solve a set of them in parallel. The first part of this thesis proposes distribution and search strategies to solve similarity queries on metric spaces, using a multi-core server with a shared memory system.

In recent years, has appeared a very promising alternative for acceleration in searching operations, it is the use of Graphics Processing Units (GPUs). Range and kNN searches provide different levels of parallelism: we can process several queries in parallel, several distance computations in parallel for a given query or even exploit parallelism in the distance operation itself. This scheme matches well with the architecture of the GPU, that execute the threads in small groups in lock-step (similar to SIMD processors). These architectures have complex memory hierarchies, and some of the memory levels can be controlled by software. Empirical studies show that it is crucial to efficiently exploit this memory system to achieve a significant performance improvement when using GPUs to accelerate a given application [48]. In the second part of this thesis, we propose strategies to map and accelerate the searching process using a single GPU NVIDIA card, and in the third part of this thesis we extended our algorithms to a hybrid multi-CPU and multi-GPU platform.

We also studied the case of processing queries in databases large enough not to fit in *device memory* (main memory of the GPU). More specifically, we implemented a hybrid algorithm which makes use of CPU-cores and GPUs in two

pipelines, and we also present a hierarchical multi-level index named *List of Super-clusters* (*LSC*), with suitable properties for memory transfer in GPU.

1.1. Objectives

The main objective of this thesis is the design, implementation and evaluation of distribution and parallel processing strategies, to process similarity queries in metric spaces on parallel platforms with a shared memory system.

The specific objectives of the present thesis are the following.

- To propose partition and distribution strategies using metric indexes to exploit parallelism using a multi-core platform.
- These latter strategies must be generic to be used with any index selected.
- To evaluate our proposals under different query traffic.
- To propose partition and distribution strategies using metric indexes on a GPU.
- To compare the performances between the multi-core and GPU strategies.
- To extend the strategies in a single GPU to a multi-GPU platform.
- To propose transfer and processing strategies to deal with databases large enough not to fit in memory.

1.2. Organization of the thesis

The following chapter of the present thesis are organized as follows.

- In Chapter 2 we provide some background on the areas of metric spaces, distributed systems, multi-thread programming, the CUDA programming model, and the relevant previous related work on these areas.

- In Chapter 3 we describe our proposals to distribute and process similarity queries using a multi-core platform. In Section 3.1 we show the common architecture used for the search strategies. In Section 3.2 we propose our strategies using different metric indexes to process similarity queries. In Section 3.3 we show the experimental results, and finally in Section 3.4 we present the main conclusions of the chapter.

- In Chapter 4 we describe our strategies to map, distribute and accelerate the searching process using a single GPU NVIDIA card. In Section 4.1 we describe our based on index proposals and experiments to process range queries, and in Section 4.2 to process *k*NN queries. In Section 4.3 we present the conclusions of the chapter.

- In Chapter 5 we extended our strategies from a single GPU to a multi-GPU platform. In Section 5.1 we study the case when the database fits in memory, and we propose and compare different distribution strategies. In Section 5.2 we study the case when the database does not fit in memory, describing our proposals and experiments using this kind of databases. In Section 5.3 we present the conclusions of the chapter.

Chapter 2

Background Knowledge and Related Work

In this chapter we describe basic knowledge about the areas of metric spaces, metric indexes, multi-core programming, GPU programming, and the related work for the present thesis.

In Section 2.1 we define the concept of metric space. All the algorithms presented in this thesis work on this kind of space. Also, we define *range* and *k*NN queries, and the increasing and decreasing methods used in sequential computing to process *k*NN queries.

In Section 2.2 we describe the different indexing methods, and the indexes that use them. Also, we describe in detail the construction and search procedures of the indexes used in this thesis, which are *EGNAT*, *M-tree*, *SSS-Index*, *SSS-Tree* and *LC*.

In Section 2.4 we describe multi-thread programming using OpenMP, and and we define the main directives, which were used in this thesis.

In Section 2.5 we present the main features of a GPU, its architecture and

memory hierarchy. Also, we show and describe an example of code using the CUDA programming model.

Finally, in Section 2.6 we expose the related work on multi-core and GPU platforms.

2.1. Metric Spaces

Searching objects from a database which are similar to a given query object is a problem that has been widely studied in recent years. The solutions are based on the use of a data structure called *index* that acts as a filter to speed up the processing of queries.

The similarity between objects can be modeled as a metric space, which is defined as follows.

A ***metric space*** [60] (X,d) is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects and holds the properties:

- Strict Positiveness: $d(x,y) > 0$ and if $d(x,y) = 0$ then $x = y$
- Symmetry: $d(x,y) = d(y,x)$
- Triangle Inequality: $d(x,z) \leq d(x,y) + d(y,z)$

The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called the database and represents the collection of objects of the search space. There are two main queries of interest, *range* and *k*NN queries.

Range Query [17]: The goal is to retrieve all the objects $u \in \mathbb{U}$ within a radius r of the query q (i.e. $(q,r)_d = \{u \in \mathbb{U} / d(q,u) \leq r\}$). See Figure 2.1(a).

The k Nearest Neighbors (kNN) [21]: The goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q,u) \leq d(q,v)$. The Figure 2.1(b) shows an illustration of a kNN query with k=5, and also is shown the radius to contain the k results. We observe that the number of possible solutions could be more than one, since there are three elements with the same distance to the query.

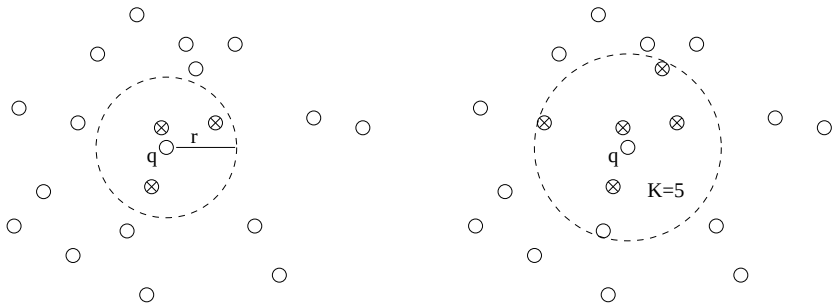


Figure 2.1: Examples of **a)** *range query* (q,r) and **b)** kNN query (q, k) with $k = 5$.

[17] shows the two main methods to solve kNN queries, and both of them use as basis the solution for range queries. They are described below.

Increasing range method: This uses an iterative range search algorithm with a predefined initial range search, increasing the range in each new iteration. The steps of the searching algorithm to solve a kNN query q are as follows. (1) Search q with an initial range $r = a^i\varepsilon$ ($a > 1$, $i = 0$, $\varepsilon \in \mathbb{R}$). (2) Increase i until (at least) k elements lies inside the search radius $r = a^i\varepsilon$.

Decreasing range method: This method decreases the range search as much as possible for each visited element. The steps of the searching algorithm to solve a range query q are as follows. (1) Search q with an initial range $r = \infty$. (2) When k

elements are found, r is adjusted to the farthest of the k elements. (3) The range is adjusted (if it is required) when a new element is visited.

2.2. Indexing

A trivial way to solve similarity queries is search exhaustively the database, which implies $O(n)$ for a database with n elements. But, the distance function is a very costly operation, thus it is not efficient to process queries on this way.

Because of the above, it is convenient to build an index from the whole database. The creation of that index must be performed off-line since its costly process. However, once built it will allow to avoid distance evaluations when solving a given query. All the indexes use the metric property triangle inequality to discard objects.

The cost given by the total time to evaluate a query can be split as:

$$T = \#\text{distance evaluations} \times \text{complexity(d)} + \text{extra CPU time} + \text{I/O time} \quad (2.1)$$

In many applications, however, evaluating $d()$ is so costly that the other components of the equation 2.1 can be neglected. Therefore, it is important to perform the minimal distance evaluations to keep a good throughput.

In order to reduce the number of distance evaluations, the indexing algorithms on metric spaces store a set of key distances. All of them partition the space X in subsets. The index allows to determine with low cost a list X_i of subsets of candidates to be part of the results. Just the non-discarded elements must be compared against the query.

To partition the database there are two main approaches: *algorithms based on pivots* and *algorithms based on clustering or compact partitions* [17].

The algorithms based on pivots preselect some key objects from the database, and these objects (or pivots) are used to filter elements using triangle inequality, avoiding to calculate the distances of these elements against the query. The searching process is as follows.

- Let $\{p_1, p_2, ..., p_N\} \in X$ be a set of pivots. For each element $x \in X$, we store in a matrix its distance to the N pivots $(d(x, p_1), ..., d(x, p_k))$. This process is off-line performed, and the matrix of distances is the structure that repesents the index.

- To process a range query (q, r), the steps to follow are:

  1. The distances from the query q to the N pivots are calculated $(d(q, p_1), ..., d(q, p_k))$.
  2. If for an element $x \in X$ is satisfied $|d(q, p_i) - d(x, p_i)| > r$, by triangle inequality we know that $d(q,x) > r$, therefore it is not necessary to calculate $d(x,q)$, and the element is discarded.
  3. All the elements that cannot be discarded by the previous step, must be compared against the query q.

Some algorithms do a direct implementation of this concept, and they differ each other in its extra structure to reduce the cost of CPU to find the candidate elements. Examples of these are: *SSS-Index* [8], *SSS-Tree* [9], *AESA* [58], *LAESA* [39], *Spaghettis* and its variants [14, 42], *FQT* and its variants [4] and *FQA* [15].

On the other hand, the algorithms based on clustering divide the space in areas, where each area has a center. Also, some relevant information about the areas is stored, to allow the discard of areas just comparing its center against the query.

There are two criteria to delimit the areas in the indexes based on clustering: *Voronoi areas* and *Covering radius*. The former divide the space using hyperplanes, and identify which of them intersect the query. The latter divide the space in balls that can be intersected and overlapped. These criteria are described as follows.

Criterion of Voronoi Partitions: A set of elements of the database are chosen as centers, so each center will belong to a different partition. Each remaining point is assigned to the partition with the nearest center. The partitions are similar to Voronoi cells in vector spaces.

- **Definition of *Voronoi Diagram***: Take a set of points $\{c_1, c_2, \ldots, c_m\}$ (centers). The Voronoi diagram is defined as the subdivision of the plane in m partitions, one per each center c_i. The query q belongs to the partition c_i iff the distance $d(q, c_i) \leq d(q, c_j)$ for each c_j, with $j \neq i$.

  During the evaluation of a range query (q, r) the distances $d(q, c_1), ..., d(q, c_m)$ are calculated. Then, the nearest center c_i is chosen, and all the partitions with center c_j that satisfy $d(q, c_j) > d(q, c_i) + 2r$ are discarded.

Criterion of Covering Radius: This divide the space in balls that can be intersected and overlapped. Each index that use this criterion defines its own algorithm to select the center of the balls. For each ball, some relevant information is stored including the center and *covering radius* of the ball. The covering radius $rc(c_i)$ is the distance between the center c_i and the farthest

element of its ball. Then, when solving a query q, the area with center c_i can be discarded if $d(q,c_i) - r > rc(c_i)$.

Some indexes combine these techniques, such as the *GNAT* [7] and *EGNAT* [41]. Another indexes just use covering radius, such as *M-trees* [18] and the *List of Clusters* [16]. Those who use Voronoi partitions are the *GHT* and its variants [55, 45], and the Voronoi-trees [23, 44].

The Figure 2.2 summarizes the indexing methods, where all of them partition the dataset into subsets. The index determines a list of subset candidates, i.e. the subsets that cannot be discarded by triangle inequality (three areas in the figure), and therefore they must be searched exhaustively.

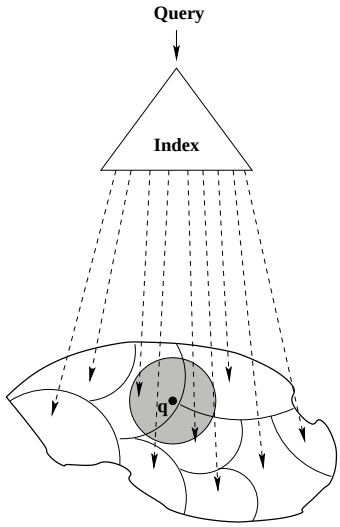


Figure 2.2: Example of indexing.

There are many metric indexes in the technique literature. In the present thesis we selected five of them, because all of them are very referenced and model the searching space in a different way, which help us to show the generality of our

algorithms. These indexes were *EGNAT* [41], *M-tree* [18], *SSS-Index* [8], *SSS-Tree* [9] and *List of Clusters* (*LC*) [16]. We describe these indexes below.

2.2.1. EGNAT

The *EGNAT* [7] index is a tree based structure optimized for secondary memory. Also, it implements a deleting method, and has two types of nodes: *gnat nodes* (inner nodes), and *bucket nodes* (leaf nodes). The elements are stored in both gnat and bucket nodes.

In the construction of the *EGNAT*, the root of the tree is composed by N random elements (c_1, c_2, c_N) called *centers*, and each center represents a different area of the space. Each remaining element will be added to the subtree of its closest center. Each subtree is recursively partitioned.

The construction algorithm of the *EGNAT* index is as follows:

1. N elements are randomly selected, called *centers*.
2. Each remaining element is associated with its closest center. The set of elements associated with the center c_i is called D_{c_i}.
3. For all couple of centers (c_i, c_j) are calculated and stored three data: (1) the range $range\left(p_i, D_{p_j}\right) = \left[min\_d\left\{\left(p_i, D_{p_j}\right)\right\}, max\_d\left\{\left(p_i, D_{p_j}\right)\right\}\right]$, and the (2) minimum and (3) maximum distances $d(c_i, x)$, where $x \in D_{c_j} \cup \left\{c_j\right\}$.
4. The previous steps are recursively applied over each subset D_{c_i}.

Each set D_{c_i} represents a subtree where the root is c_i, i.e. each D_{c_i} corresponds with the Voronoi region whose center is c_i. The Figure 2.3 shows an example of

construction of the first level of the *EGNAT* with $N = 4$, and it also shows the range table that must be stored for each center c_i. In the example of the figure, the elements were inserted in the order of its numeric value. If a new object c_{16} is inserted, and if that object is assigned to the area where c_4 is center, then a new node would be created and it would increase the level of the tree, just as is shown by Figure 2.4.

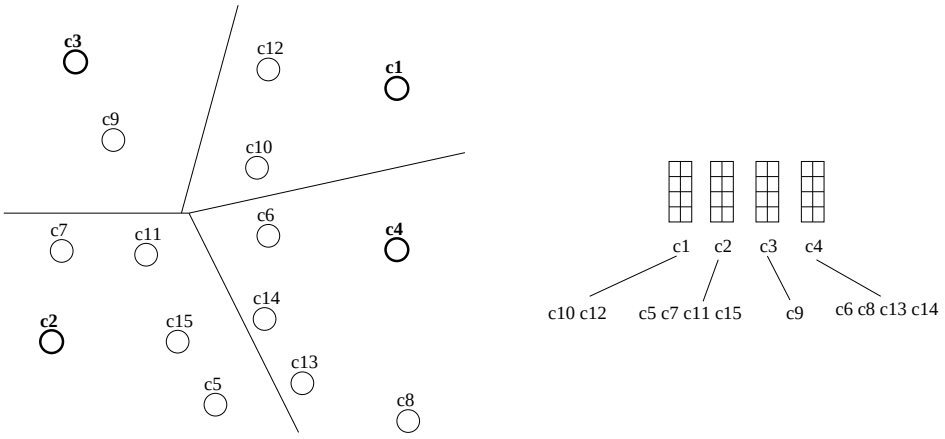


Figure 2.3: *EGNAT*: Construction of the tree.

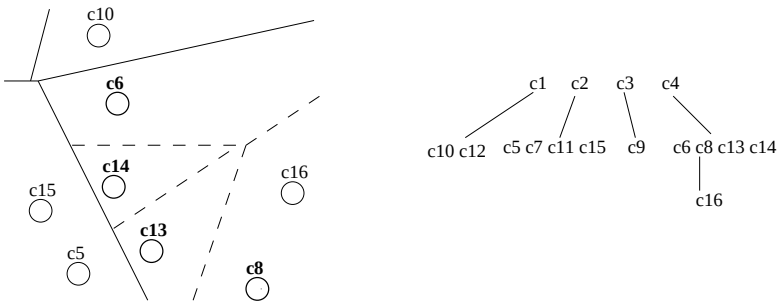


Figure 2.4: *EGNAT*: Insertion of a new object.

The Algorithm 1 shows the search for a range query (q, r) with the *EGNAT*. The search is recursively applied as follows.

1. We assume that we are interested in retrieving all objects with distance $d \leq r$ to the range query (q,r). Let C be the set of centers of the current node in the search tree. The first time C is the root.

2. An element $c \in C$ is randomly selected, and the distance $d(q,c)$ is calculated. If $d(q,c) \leq r$, add c to the output set result.

3. $\forall x \in C$, if $[d(q,c)-r, d(q,c)+r] \cap range(c,D_x)$ is not empty, x is added to C'.

4. For all element $c_i \in C'$, repeat recursively the search in D_{c_i}.

Algorithm 1 *EGNAT*: range search algorithm.

```
range_search(Node C, Query q, Range r)
 1: {Let Results be the result set}
 2: R ← ∅
 3: c_r ← random_element(C)
 4: d ← dist(c_r, q)
 5: if d <= r then
 6:     Results.add(c_r)
 7: end if
 8: range(c_r, q) ← [d − r, d + r]
 9: for all x ∈ C do
10:     if range(c_r, q) ∩ range(c_r, D_x) ≠ ∅ then
11:         C'.add(x)
12:         if distance(x, q) <= r then
13:             Results.add(x)
14:         end if
15:     end if
16: end for
17: for all c_i ∈ C' do
18:     range_search(D_{c_i}, q, r)
19: end for
```

The reason to be able to discard subtrees in the step 3 of the search is illustrated in the Figure 2.5, and it is as follows: Let y be an element in D_x. If $d(y,c) < d(q,c) - r$, then we have by triangle inequality that $d(q,y) + d(y,c) \geq d(q,c)$, thus $d(q,y) > r$. In the same way, if $d(y,c) > d(q,c) + r$, we use triangle inequality in $d(y,q) + d(q,c) \geq d(y,c)$ to deduce $d(q,y) > r$.

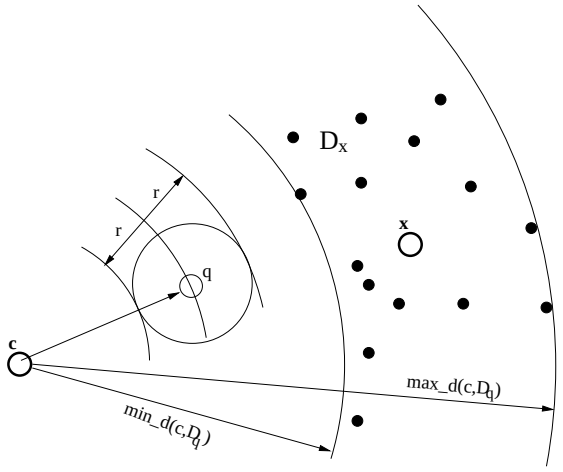


Figure 2.5: Discard of subtrees using ranges. D_x is discarded because $d(q,c)+r < min\_d(c,D_x)$.

2.2.2. M-Tree

The *M-Tree* index [18] was the first dynamic structure that allowed delete and reinsertion. It is a balanced tree, with fixed size of node. It is similar to a B-Tree and evolves in a bottom-up way. The public version of the M-Tree ([3]) was not available in C language, but for fair comparison we implemented it in C, and we made it public in [1].

This index is created by selecting randomly a set of elements as centers (these first centers will compose the root node), and each center stores its *covering radius*. The covering radius is the distance between the center and the farthest element assigned to it. Each remaining element is inserted in the most suitable subtree, which is defined as the one that increase less its covering radius. In case of overflowing of a node, we use the methods *mM_RAD* and *Generalized Hyperplane*, which combination showed the best results in [18]. *mM_RAD* selects which pair

of elements (O_1 and O_2) will be promoted to the parent node, which are those that increase less the covering radius. After O_1 and O_2 are promoted, *Generalized Hyperplane* calculates for each remaining element (O_j) of the node, the distances $d_1 = d(O_1, O_j)$ and $d_2 = d(O_2, O_j)$, then if $d_1 < d_2$ the element O_j is assigned to the node linked by O_1, or to that linked by O_2 in the opposite case.

The inner nodes of the M-Tree are different from the leaf nodes. The former store the route of the objects and are called *routing objects*, and the latter store the elements of the database.

The information stored for a routing object is:

- O_r: routing object.
- $T(O_r)$: subtree of O_r.
- $ptr(T(O_r))$: pointer to the root of $T(O_r)$.
- $rc(O_r)$: covering radius of O_r.
- $d(O_r, P(O_r))$: distance from O_r to its parent.

The information stored for a leaf node is:

- O_j: Stored element.
- $Oid(O_j)$: ID of the object O_j.
- $d(O_j, P(O_j))$: distance from O_j to its parent.

In the Figure 2.6 we show an illustration of the structure of a M-Tree and the covering radii of each routing object.

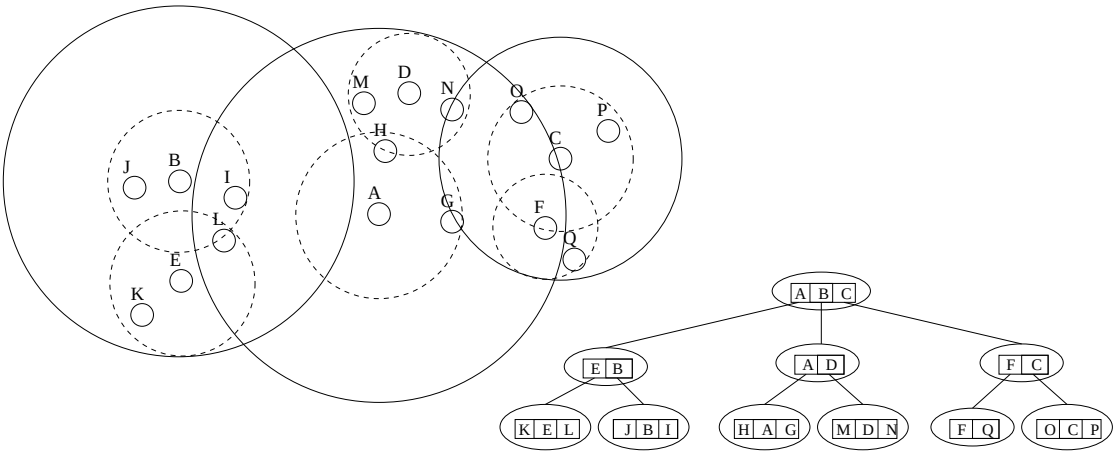


Figure 2.6: *M-tree*: Example of structure and its representation in a 2-dimensional space.

The algorithm 2 shows the range search of the M-Tree. It begins with the root node and recursively are visited all the non-discarded (by triangle inequality) children.

Algorithm 2 *M-Tree*: range search algorithm.

range_search(Node N, Query q, Range r)

1: {Let O_p be the parent object of the node N}
2: **if** N is a routing object **then**
3: **for all** $O_r \in N$ **do**
4: **if** $\left|d(O_p,q) - d(O_r,O_p)\right| <= r + rc(O_r)$ **then**
5: **if** $distance(O_r,q) <= r + rc(O_r)$ **then**
6: range_search($ptr(T(O_r)),q,r$)
7: **end if**
8: **end if**
9: **end for**
10: **else**
11: **for all** $O_j \in N$ **do**
12: **if** $\left|d(O_p,q) - d(O_j,O_p)\right| <= r$ **then**
13: **if** $distance(O_j,q) <= r$ **then**
14: $Results.add(oid(O_j))$
15: **end if**
16: **end if**
17: **end for**
18: **end if**

2.2.3. Sparse Spatial Selection (SSS-Index)

During construction, this pivot-based index [8] selects some objects as *pivots* from the collection. The Algorithm 3 shows the pivot selection algorithm, which works as follows. Let $(\mathbb{X}, d)$ be a metric space, $\mathbb{U} \subset \mathbb{X}$ an object collection, and M the maximum distance between any pair of objects, $M = max\{d(x,y)/x,y \in \mathbb{U}\}$. The set of pivots contains initially only the first (random) object ($u_1 \in \mathbb{U}$) of the collection. Then, for each remaining element $u_i \in \mathbb{U}$, u_i is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than αM, being α a constant parameter. Therefore, an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots.

Algorithm 3 *SSS-Index*: pivot selection algorithm.

```
1: {Let U be the database}
2: {Let M be the maximum possible distance between two element of U}
3: {Let α be a real constant}
4: PIVOTES ← {u_1}
5: for all u_i ∈ U do
6:     if ∀p ∈ PIVOTES, distance(u_i, p) >= Mα then
7:         PIVOTES ← PIVOTES ⋃ {u_i}
8:     end if
9: end for
```

During the creation process, a table of distances is created, where columns are the pivots and rows the objects. Each cell in the table contains the distance between the object and the respective pivot. The distances of the table are calculated in a off-line way.

The Algorithm 4 shows the range search algorithm of the *SSS-Index*, which works as follows. For a range query (q,r) the distances between the query and all pivots are computed (line 5). An object u from the collection can be discarded if

there exists a pivot p_i for which the condition $|d(p_i,u) - d(p_i,q)| > r$ does hold (line 11). The objects that pass this test are considered as potential members of the final set of objects that form part of the solution for the query and therefore they are directly compared against the query by applying the condition $d(u,q) \leq r$ (line 17).

Algorithm 4 *SSS-Index*: range search algorithm.

```
range_search(Query q, Range r)
 1: {Let U be the database}
 2: {Let DIST be the distance table}
 3: {Let PIVOTS be the set of pivots}
 4: i = 0;
 5: for all u_i ∈ PIVOTS do
 6:     arrD[i++] = distance(u_i, q)
 7: end for
 8: for i = 0; i < U.size(); i++ do
 9:     discarded = false
10:     for j = 0; j < PIVOTS.size(); j++ do
11:         if arrD[j] < DIST[i][j] - r || arrD[j] > DIST[i][j] + r then
12:             discarded = true
13:             break
14:         end if
15:     end for
16:     if !discarded then
17:         if distance(U[i], q) <= r then
18:             Results.add(U[i])
19:         end if
20:     end if
21: end for
```

2.2.4. SSS-Tree

The *SSS-Tree* index [9] is based on a tree structure, that use clustering to divide the space. The cluster center of each inner node is chosen by the SSS technique (Algorithm 3).

The construction procedure begins with all the elements inside one *bucket node*. Let D be the set of elements of this bucket node, M the maximum distance between two elements of D, and α a real constant, then we perform the SSS procedure on

D. This means, the first element of D is selected as a pivot, then for each remaining element $u_i \in D$: if for all pivot p, $d(u_i, p) >= M\alpha$, then u_i is a new pivot, otherwise it is assigned to the cluster of the nearest pivot. For the latter, each pivot has a pointer to a cluster of elements. The Figure 2.7 illustrates an example of an SSS-Tree index after selecting the pivots corresponding to the initial *bucket node*. Then, the process is applied over each cluster recursively. The recursion ends when the quantity of elements of the cluster is less or equal to a constant β.

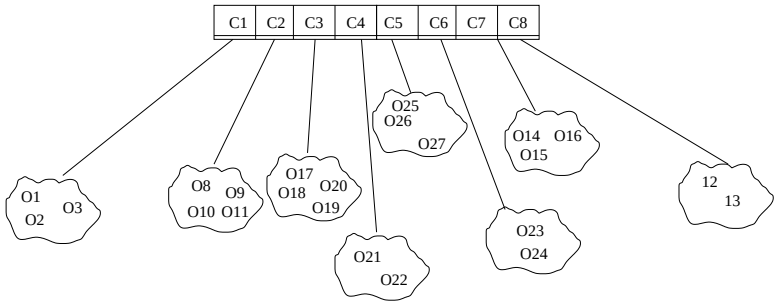


Figure 2.7: *SSS-Tree*: Structure after first step of construction.

The inner and leaf nodes have the same structure with the fields described as follows.

- **info:** Element of the database.
- **type:** Type of node (inner or leaf).
- **covering_radius:** Set of maximum distances between each center of the node and the elements of its subtree.
- **Pchild:** Pointers to the child nodes.
- **Nchild:** Number of elements of the child nodes.

The Algorithm 5 shows the range search of the SSS-Tree index, where the covering radius is used to discard subtrees by triangle inequality.

Algorithm 5 *SSS-Tree*: range search algorithm.

```
range_search(Node N, Query q, Range r)
 1: {Let c be a center of the node N}
 2: for i = 0; i < N.size(); i++ do
 3:     dist = distance(c.info, q)
 4:     if dist <= r then
 5:         Results.add(c.info)
 6:     end if
 7:     if c.Pchild != NULL then
 8:         if dist − r <= c.covering_radius then
 9:             range_search(c.Pchild, q, r);
10:         end if
11:     end if
12: end for
```

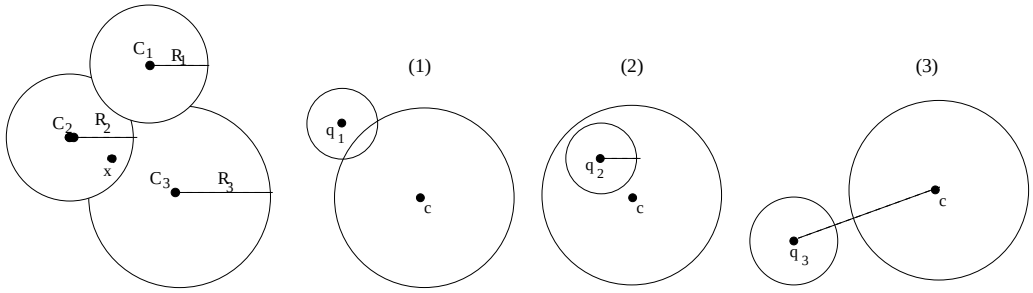
2.2.5. List of Clusters (LC)

The *List of Cluster* (*LC*) [16] can be implemented dividing the space in two different ways: taking a fixed radius for each partition or using a fixed size. In this thesis, to ensure good load balance in the parallel platform, we consider partitions with a fixed size of B elements, thus the radius rc of a cluster with center C delimit the set of the B nearest elements ($\in \mathbb{U}$) from C ($k\text{NN}_{\mathbb{U}}(C,K)$).

The *LC* data structure is created from a set of centers (objects). The construction procedure (illustrated in Figure 2.8(a)) is roughly as follows. We (randomly) chose an object $C_1 \in \mathbb{U}$ which becomes the first center. This center determines a cluster (C_1, R_1, I_1) where I_1 is the set $k\text{NN}_{\mathbb{U}}(C_1,B)$ of B nearest neighbors of C_1 in $\mathbb{U}$ and R_1 is the distance between the center C_1 and the farthest element in I_1 (R_1 is called *covering radius*). Next, we choose a second center C_2 from the set $E_1 = \mathbb{U} - (I_1 \cup \{C_1\})$. This second center C_2 determines a new cluster (C_2, R_2, I_2) where I_2 is the

set $k\text{NN}_{E_1}(C_2, B)$ of B nearest neighbors of C_2 in E_1 and R_2 is the distance between the center C_2 and the farthest element in I_2. Let $E_0 = \mathbb{U}$, the process continues in the same way choosing each center C_n ($n > 2$) from the set $E_{n-1} = E_{n-2} - (I_{n-1} \cup \{C_{n-1}\})$, till E_{n-1} is empty.

Note that, a cluster created first during construction has preference over the following ones when their corresponding covering radius overlap. All the elements that lie inside the cluster corresponding to the first center C_1 are stored in it, despite that they may also lie inside the subsequent clusters. This fact is reflected in the search procedure. Figure 2.8(b) illustrates all the situations that can occur between a range query (q, r) and a cluster with center C and covering radius r_c.



(a) The influence zones of three centers taken in the order C_1, C_2 and C_3.

(b) For q_1 we have to consider the current cluster and the rest of the centers. For q_2 we consider the current cluster and we can stop the search avoiding the remaining centers. For q_3 we can avoid considering the current cluster.

Figure 2.8: **a)** Example of construction of *LC* and **b)** cases of searching.

During the processing of a range query (q, r), the idea is that if the first cluster is (C_1, R_1, I_1), we evaluate $d(q, C_1)$ and add C_1 to the result set if $d(q, C_1) \leq r$. Then, we scan exhaustively the objects in I_1 only if the range query (q, r) intersects the ball with center C_1 and radius R_1, i.e. only if $d(q, C_1) \leq R_1 + r$ (case 1 in Fig. 2.8(b)). Next, we continue with the remaining set of clusters following the construction order. However, if a range query (q, r) is totally contained in a cluster

(C_i, R_i, I_i), i.e., if $d(q, C_i) \leq R_i - r$ (case 2 in Fig. 2.8(b)), we do not need to traverse the remaining clusters, since the construction process of the *LC* ensures that all the elements that are inside the query (q, r) have been inserted in I_i or in a previous clusters in the building order. In [16] different heuristics have been presented to select the centers, and it has been experimentally shown that the best strategy is to choose the next center as the element that maximizes the sum of distances to previous centers. Thus, in this work we use this heuristic to select the centers.

2.3. High Dimensional Spaces

Although all the indexes and methods above are efficient discarding elements, reducing the number of distance evaluations and reducing running time, all of them lose efficiency in *high dimensional spaces*. The traditional indexing techniques for metric spaces have a exponential dependence with the dimension of the space, i.e. when the dimension increase, those techniques become less efficient. The reason is because when the dimension grows, the capacity of dividing the searching space is reduced.

In high dimensional spaces the quantity of elements that can be discarded is very low. In [17] is defined the concept of *intrinsic dimensionality* of a metric space as $\rho = \frac{\mu^2}{2\sigma^2}$, where μ and σ^2 are the mean and variance of the histogram of distances between elements of the same space. Thus, a high intrinsic dimensional space has a concentrated histogram of distances, i.e. the mean is high and the variance low, which indicates that all the elements are very close between them.

2.4. Multi-core systems and OpenMP

Nowadays the computer architectures that include several cores in one single CPU are more common, and also there are several libraries that allows multi-thread programming. With this we are able to execute threads in parallel, running on a different core.

There exist several models and standards that allow multi-thread programming, such as, Pthreads([43]), TBB ([51]) u OpenMP ([12]). For the purpose of this thesis, we selected the OpenMP library. This was mainly because its high level of abstraction, due to the resulting code is similar to the sequential one, which is very useful to develop and debug programs. OpenMP has been developed from 1997, therefore the current compilers are a quite robust. Also, the fact of the team that manages OpenMP is composed by different companies (AMD, Intel, Sun Mycrosystems and others) and not of just one, provides confidence.

OpenMP is organized by *directives* and functions, such as:

Functions:

`omp_set_num_threads(`P`):` Set the number of threads of the program (P threads in this case).

`omp_get_thread_num():` Returns the ID of the thread.

`omp_get_num_threads():` Returns the quantity of threads that are currently running.

`omp_get_num_procs():` Returns the available quantity of cores in the system.

Directives:

`#pragma omp parallel`: In this point of the code are created P threads, and all of them execute the same code, but each thread has an unique ID. Also, this directive sets which variables will be global or private for the threads.

`#pragma omp critical`: This directive allows to set a code zone as critical, which means that code will be executed for just one thread at a time.

`#pragma omp master`: This directive set a zone code that will be executed for just the master thread (thread with $ID = 0$).

`#pragma omp barrier`: When a thread reaches this directive, stops its execution until all the rest of threads reach this directive too.

The Figure 2.9 shows a program example using OpenMP. In this figure, the directive `#pragma omp critical` solves the concurrency problem allowing the access to the *global* variable to just one thread at a time. The directive `#pragma omp barrier` guarantees that when the master thread prints the *global* variable all the access to it has been already done.

We also distribute the threads among the cores using the *sched.h* library. Always a thread is assigned to a different core from the rest, aiming to avoid conflict of resources in the same core.

```
#include <stdio.h>
#include <omp.h>
main ()
{
   int id_private, global=0;
   omp_set_num_threads(4);/* We used 4 Threads */
   #pragma omp parallel shared(global) private(id_private)
   {
      /* In this point are created 4 threads.
         All of them execute the following code */
      id_private = omp_get_thread_num();
      printf("I am the thread with ID=%d\n", id_private);
      #pragma omp critical
      {
         global++;
      }
      #pragma omp barrier
      #pragma omp master
      {
         printf("Global variable = %d", global);
      }
    }  /* end of the parallel block */
    return 0;
}
```

Output:

```
I am the thread with ID=0
I am the thread with ID=2
I am the thread with ID=1
I am the thread with ID=3
Global variable = 4
```

Figure 2.9: Program example using OpenMP.

2.5. Graphic Process Unit (GPU)

This section presents an overview of the architecture used by NVIDIA's GPUs [47] and the programming model offered by their CUDA drivers, in order to expose the challenges that compilers have to face to produce efficient codes for these devices.

A GPU is a device that can be used as a high performance coprocessor, suitable for accelerating data parallel codes. The program running on the CPU (the host) must explicitly manage data transfers from host memory to *device memory* and vice versa, and can control the execution of programs on the device.

Figure 2.10 gives an schematic view of the actual hardware of modern NVIDIA's GPU. The GPU cores (processors) are organized into several *multiprocessors*. Each of these cores integrates its own functional units and a large register file that accommodates the execution of hundreds of concurrent threads – to tolerate the long latency associated with the accesses to the graphic's board memory –. The multiprocessors integrate a single instruction unit and a local shared memory [1]. The memory hierarchy also includes read-only cache memories to speed up access to textures and constants, which are shared by pairs of multiprocessors. The CUDA Block abstraction is closely related to this organization: each CUDA Block is executed by one multiprocessor, which depending on the resource availability can accommodate multiple blocks concurrently – the actual number of concurrent threads per multiprocessor is limited by the allocated resources –.

Each multiprocessor can maintain hundreds of threads in execution. These threads are organized in sets, called *warps*[2]. Every cycle, the hardware scheduler of each multiprocessor chooses the next warp to execute (i.e. no individual threads but warps are swapped in and out), using fine grain simultaneous multithreading to hide memory access latencies. This execution model is called Single Instruction Multiple Thread (SIMT) by Nvidia.

Regarding the memory hierarchy, all multiprocessors can access the same on-board DRAM memory (global memory in CUDA parlance) through a high bandwidth bus. This global memory is banked, which allows the hardware to coalesce several simultaneous memory accesses to adjacent positions into a single memory transaction. In addition, each multiprocessor contains a smaller SRAM memory.

[1] On the GT200 series, there is also a single double-precision unit per multiprocessor

[2] The size of a warp is currently 32

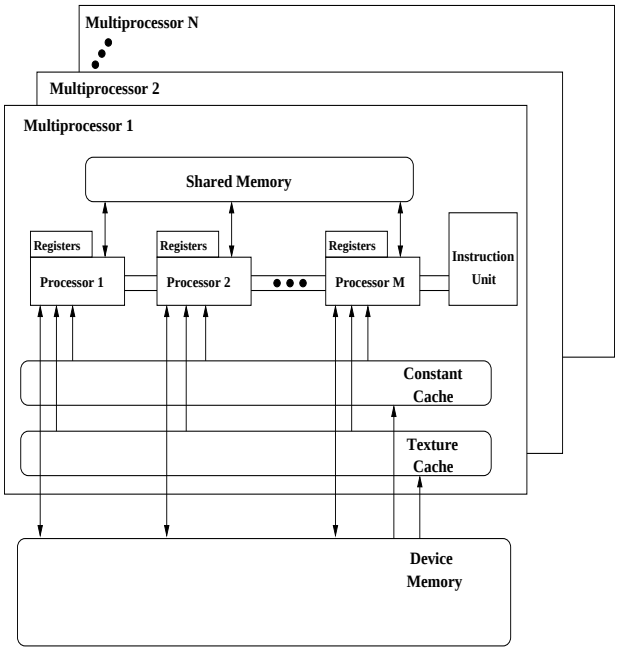


Figure 2.10: The CUDA programming model is designed for compute. It represents the GPU as a coprocessor that integrates several multiprocessors and a complex memory hierarchy.

In more recent GPUs (starting from the *Fermi* architecture [47]) this SRAM can be configured as scratch pad (i.e. a software controlled memory) and hardware controlled cache memory. The user can decide, with certain restrictions, the amount of cache and scratch pad needed. These newer GPUs also incorporate a L2 cache common to all multiprocessors. Finally, GPU multiprocessors can also access the global memory through a special read-only two level hierarchy of so called *texture* caches, that can be configured to capture 2D locality.

This model is exposed to the programmer by the CUDA driver. It allows to control the execution of a *kernel* on the device. A *kernel* consists of a sequential piece of code that has to be executed by a large set of threads on the GPU multiprocessors. Threads within a warp are simultaneously executed on the scalar processors of a

single multiprocessor in lock step. If the threads in a warp execute different code paths, only those that follow the same path can be executed simultaneously and a penalty is incurred.

Warps are further organized into a grid of *CUDA Blocks*: threads within a CUDA Block are all executed in the same multiprocessor, and are then able to cooperate with each other by (1) efficiently sharing data through the shared low latency local SRAM memory and by, (2) synchronizing their execution via barriers. In contrast, threads from different CUDA Blocks can be (potentially) scheduled on different multiprocessors and thus they can only coordinate their execution via accesses to the high latency global memory. Within certain restrictions, the programmer specifies how many CUDA Blocks and how many threads per CUDA Block are assigned to the execution of a given kernel. When a kernel is launched, threads are created by hardware and dispatched to the GPU cores.

According to NVIDIA, the most significant factor affecting performance is the bandwidth usage. Although the GPU takes advantage of multithreading to hide memory access latencies, having hundreds of threads simultaneously accessing the global memory introduces a high pressure on the memory bus bandwidth. Therefore, reducing global memory accesses, by using local shared memory to exploit inter thread locality and data reuse, largely improves kernel execution time. In addition, improving memory access patterns is important to allow coalescing of warp-wise memory accesses and to avoid bank conflicts on shared memory accesses.

To summarize, we can draw some conclusions on the challenges being faced when mapping code to this kind of devices. First the programmer needs to partition the code into host and GPU code. The parallel code pieces for the GPU must be mapped onto the CUDA model of blocks and threads. Here we have two differ-

ent levels of parallelism, independent threads that are assigned to different CUDA Blocks and cooperating threads, that are assigned to the same CUDA Block forming warps. The latter should exhibit SIMD parallelism to avoid warp divergences and to minimize the number of non-coalesced memory accesses (threads in the same warp should access adjacent memory addresses). In addition, to reduce bandwidth requirements, data locality should be efficiently exploited in the register file and the local shared memories. This implies that the programmer should explicitly consider, schedule and express data transfers between the different memories available[3], trying to reduce the accesses to the global memory. The existence of the new caches introduces an additional variable that should be considered.

The Figure 2.11 shows a code that sum two vectors in GPU. To do it, a space in device memory is allocated for variables (using `cudaMalloc`), then these variables are initialized (using `cudaMemcpy`). A kernel is launched with 1 CUDA Block and 200 threads per CUDA Block. Each thread gets its ID (tid variable) using predefined variables for the GPU, ID_{Thread} (represents the ID of the thread into the CUDA Block), T_{Block} (represents the quantity of threads per CUDA Block), and ID_{Block} (represents the ID of the CUDA Block). Each thread sum the i-th of both arrays (dev_A and dev_B). When the kernel is done, the data is copied to the CPU to print the results.

[3]This programmer control is larger when using the SRAM mainly as a software controlled memory, but hardware controlled cache must also be taken into account during the mapping

```
#include <stdio.h>
#include <cuda.h>
#define N 200
#define N_BLOQUES 1

/* We define a kernel called 'function' */
__global__ void function(float *A, float *B, float *result)
{
   /* We get the ID of the thread */
     int tid = ID_Thread + ( T_Block * ID_Block );
   /* Each thread sum a different thread */
     result [tid] = A[tid] + B[tid];
     return;
}

main()
{
     float host_A[N], host_B[N], host_result[N];
     float *dev_A, *dev_B, *dev_result;

  /* We allocate memory in device memory */
     cudaMalloc((void **)&dev_A, sizeof(float)*N);
     cudaMalloc((void **)&dev_B, sizeof(float)*N);
     cudaMalloc((void **)&dev_result, sizeof(float)*N);

     initialize_values(host_A, host_B, host_result);
  /* We copy the values to the variables allocated in device memory */
     cudaMemcpy(dev_A, host_A, sizeof(float)*N, cudaMemcpyHostToDevice);
     cudaMemcpy(dev_B, host_B, sizeof(float)*N, cudaMemcpyHostToDevice);

  /* We invoke a kernel with 'N_BLOQUES' blocks and 'N' threads per block
     with the variables allocated in device memory */
     function<<<N_BLOQUES, N>>>(dev_A, dev_B, dev_result);

  /* We copy the results to CPU mempry */
    cudaMemcpy(host_result, dev_result, sizeof(float)*N,
               cudaMemcpyDeviceToHost);
    print_results(host_result);
    cudaThreadExit();
    return 0;
}
```

Figure 2.11: Example of summing vectors with CUDA.

2.6. Related Work about Parallelism on Metric Spaces

In the following sections we show the related work regarding to parallelism on metric spaces. In Section 2.6.1 we show the related work about publications that use distributed memory parallel platforms to accelerate search on metric spaces. In Section 2.6.2 we show the related work about GPUs used to solve similarity queries, with brute force and indexing methods.

2.6.1. Related Work on multi-core Systems

To our knowledge, there is not related work about acceleration algorithms using metric indexes with a shared memory system. Therefore, we used as basis of the Chapter 3, publications about parallelism on distributed memory systems that use metric indexes, which are [37, 19, 27, 20, 33, 35, 36, 56]. We describe them below.

[37] also uses as basis the *LC* index, and a distributed memory cluster with 120 dual-core processors. The authors propose two different distributions with the aim of enabling efficient similarity search in large-scale Web search engines. The first one makes use of specific knowledge about the workload generated by user queries. The second one proposes to disregard user behavior to look instead at the relationships among the index clusters themselves to decide their placement onto processors. Both methods perform efficiently depending on the context and they are generic enough to be applied to different distributed index data structures for metric space databases.

[19] uses the *SSS-Index* (Section 2.2.3), and a distributed memory cluster of 32 nodes (CPUs) with the BSP parallel programming model [57]. They propose several strategies for the construction and search on the index. The best proposed

construction strategy distributes the elements of the database in a circular manner among the nodes, and each node calculates the pivots using its local data with a global M (maximum distance between two elements of the database), and then all the nodes distribute its local pivots to all the rest, therefore all the nodes will have the same pivots. The best proposed search strategy sends the query to just one node of the cluster, and this node is called *ranker* of the query. After that, the ranker performs a broadcast of the query to all the nodes, and each node return to the ranker the results found on its local data. Finally, the ranker selects the final results.

[27] uses as basis the *LC* index, and a distributed memory cluster with 200 dual-core processors. They propose and compare a set of seven different distribution strategies of the index among the processors. The distributions are local, global and combinations of them. Each strategy is able to achieve efficient performance and each one is suitable for a given trade-off between performance and memory space.

[20] also uses the *LC* index, and a distributed memory cluster of 32 nodes, using the parallel libraries BSP and PVM. The main conclusions show that: (1) A global distribution of the index outperforms a local one. (2) Under a synchronous system is more suitable to use big batches of queries to take advantage of the bulk feature of the BSP model, but under an asynchronous system to use small batches of queries show better results. (3) The parallel system to be used is determined by the query traffic.

[33] uses the SA-Tree index ([40]), which is based on a tree structure. They use a distributed memory cluster of 4 nodes, and the distribution strategy with best results was a global distribution. This latter distribute the subtrees of the index among the processors, but using a limit on the distance evaluations performed in each superstep of the BSP model.

[35] proposes an index which is a combination between the *LC* and *SSS-Index*, using a distributed memory cluster of 32 nodes. Also, they propose different parallel strategies to be used with the index, in synchronous and asynchronous mode, using the BSP and MPI parallel libraries respectively. The best strategy was a hybrid algorithm which is able to change from one parallel mode to another depending on the query traffic.

[36] and [56] use the *EGNAT* index ([41]), which is based on a unbalanced tree, using a distributed memory cluster of 10 nodes. They proposed local and global distributions of the index. As conclusion, the local distribution outperformed the global one. The features of this index, such as the imbalance of the tree benefits a local approach. In the local distribution the elements are distributed among the nodes, and each node creates its local index with its local data.

In summary, the distribution strategies can be classified as *local* or *global* approaches. One of the main features of the local strategies is that each processor (or node) can process a query independently and without communication with the rest, and then transfer the results to the broker processor (processor in charge of presenting the results to the user). A typical example of local strategy is as follows: There are a cluster of P processors and a dataset of N elements. The data are distributed among the processors, thus each processor has N/P elements. Then, each processor create its own local index with its local assigned data, and therefore each index will be one part of the whole database. The above implies that the query must be processed by all the processors. The final result is the union of the partial results of each processor.

Local strategies have the advantage of not using communication resources with the rest, but has the disadvantage of all the processors must process the same query.

On the other hand, the global strategies have just one common index for all the processors, and each query is solved using R processors, where $1 <= R <= P$. A global distribution must define an algorithm to distribute the elements of the global index among the processors. The specific features of an index can be a deciding factor to choose a local or global distribution.

2.6.2. Related Work on GPU

The related work about the use of GPUs with metric indexes to accelerate the search process on metric spaces is few. But, they can be classified according to the type of query that solve. The publication that give a solution to the range query is [49], and those that give a solution to kNN queries are [30], [26] and [10]. We describe them below.

[49] proposes a solution for range queries on a single GPU platform using the *Spaghettis* metric index [14]. This index is based on pivots, and in the experiments is just used a database of words. The authors propose a CUDA based algorithm of three steps: (1) In the first step, the distances from the queries to all the pivots are calculated. One kernel is launched with as many threads as number of queries, and each thread calculates the distances between a query and all the pivots. (2) In the second step, the candidates to be part of the results are found. As many kernels as number of queries are launched, and each kernel is launched with as many threads as number of elements in the database. Each kernel in this step returns the candidates for a given query, where each thread try to discard (using the metric property of triangle inequality) a different element using the distances of the first step to all the pivots. (3) The third step calculates the distances between the candidates and

the query. One kernel is launched with as many threads as candidates for each query. Each thread calculates the distance between a candidate and a query, and determines if the candidate is or not a solution. They compare its *Spaghettis* index on GPU against the sequential version of the index, and also against a brute force algorithm in GPU, obtaining a speed-up up to 9.8x.

[30] proposes a solution for *k*NN queries on a single GPU platform. The authors propose to divide the database of elements (matrix A of dimension $n \times d$) and queries (matrix B of dimension $m \times d$) in small submatrices of size $T \times T$ (Figure 2.12). Each CUDA Block loads submatrices of A and B in shared memory to write the result submatrix in C. This implies that each CUDA Block will perform $\left(\frac{d}{T}\right)^2 T^2 = d^2$ reads to device memory, where T is a constant limited by the size restrictions of shared memory, and the number of required CUDA Blocks is $(m/T) \times (n/T)$. Then, the result distances matrix is sorted using the *CUDA-based Radix Sort* [53], and finally are chosen the first K elements as final results.

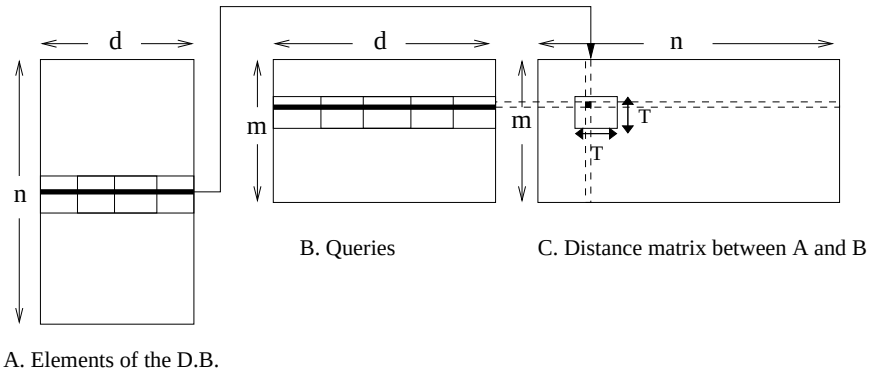


Figure 2.12: Partitioning of the data and query matrices in submatrices.

[26] proposes a brute force method to solve *k*NN queries using a single GPU. It calculates the distances between all the elements of the database and the query, then

store the results in an array of distances d, and finally sort the array d to present the first K elements of d as the final results. They compare two sorting algorithm implemented for GPU in the same work, *comb sort* and *insertion sort*, where the latter showed a better performance.

[10] proposes a solution for kNN queries using a single GPU just taking account the particular case of $k = 1$, and the proposal is based on the management of texture memory. It calculates the distances between all the elements of the database and the query, and then those distances are reduced storing the partial results in texture memory. In the present thesis, we cover the case of kNN queries with $k \geq 1$.

Chapter 3

Distribution and Searching Strategies on a multi-core Platform

This chapter shows our proposed strategies to distribute and search on metric indexes using a multi-core platform. Our algorithms can be used with any metric index that is able to stop a search in the middle of its execution. To show the generality of our algorithms we used in this chapter five indexes, where all of them model the space in a different way. These indexes are *EGNAT* [41], *M-tree* [18], *SSS-Index* [8], *SSS-Tree* [9] and *List of Clusters* (*LC*) [16].

The following of the chapter is organized as follows. In Section 3.1 we show the common architecture used for all the strategies. In Section 3.2, we describe our proposed distribution and search strategies. In Section 3.3 we show the experimental results, and finally in Section 3.4 we describe the main conclusions of the chapter.

3.1. Search Model

In this section we present the common architecture for searching used for all the strategies proposed in the present chapter. The architecture is shown by Figure 3.1, where the queries are distributed in a circular manner among the threads by a broker machine.

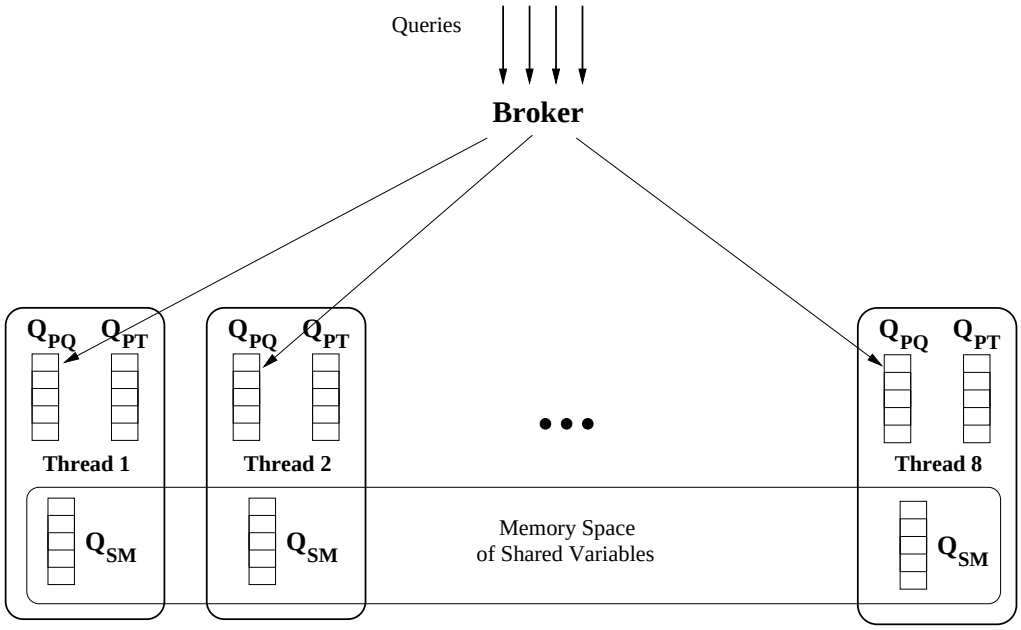


Figure 3.1: Architecture of searching.

Each thread has three queues, the first is the *Private Queue of Queries* (Q_{PQ}), where the queries arrive from the broker. The second queue is the *Private Queue of Tasks* (Q_{PT}), where the *tasks* to be processed are stored. A *task* is a structure with the necessary information to perform certain quantity of distance evaluations. The third is the *Shared Queue of Messages* (Q_{SM}), where the messages addressed for other threads are stored.

The queues Q_{PQ} and Q_{PT} are private to each thread, but in order to implement message communication Q_{SM} is public, i.e. all the threads have access to all the Q_{SM}. Due to the size variation of the Q_{PT} queue, its elements are dynamically created.

3.2. Description of the Search

All the search strategies proposed in this chapter are based on the processing of *tasks*. When a query is taken from Q_{PQ}, an initial task is generated and it is added to Q_{PT} to begin its search process. Depending on the index, the processing of a task can generate more tasks, and the process of the new ones can generate more, and so on until complete all the tasks required by the query.

The structure of a task is composed by the following fields:

region Indicates the region of the index that must be accessed to process the query. That region depends on the index. In the case of the indexes based on tree (*M-Tree*, *SSS-Tree* y *EGNAT*) this field represents a node of the tree. In the case of the *SSS-Index* this field represents an element of the database and all its distances to the pivots. In the *LC* this field represents a cluster.

index It is the element of the node where the search must begins. This is used to resume an interrupted search due to reach *R* distance evaluations.

query The query itself.

extra Some extra information specific for the index.

Thereby each task implies to perform L distance evaluations, where $1 <= L <= N_{region}$ (N_{region} is the quantity of elements stored in the memory, pointed by the field *region*). In the case of the indexes based on tree N_{region} is the quantity of elements of a node. The Figure 3.2 shows the path when processing a query in the case of an index based on tree. In this case a task cover one node of the tree, i.e. processing a task in this case implies processing one node. In the figure, when the initial task (node 1) is processed, it generates two new tasks, which implies to process the nodes 2 and 4, and these new tasks generate more again. The process finishes when all the generated tasks (in this case 6 tasks) are solved.

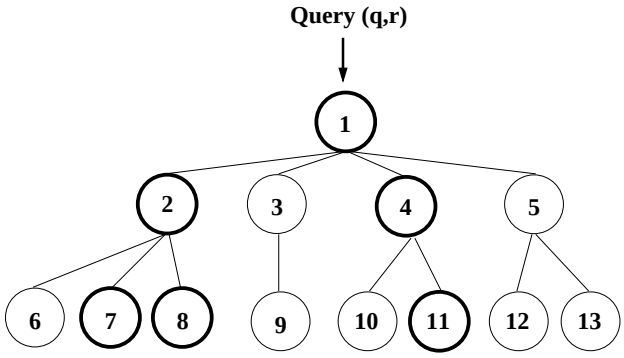


Figure 3.2: Nodes involved in the solution of a range query (q,r). In this case of an index based on tree, to process a task implies to process one node.

For all the following strategies, the queries are always distributed in a circular manner, and also because the results shown in Section 3.3.2, in all the following strategies the index is not partitioned, i.e. there is just one global index in memory, which is accessed by all the threads. All the threads access to this same shared global index to solve its queries, which means that several threads can be accessing to the same address memory at the same time for reading data. In the following, we described the multi-core strategies that have been explored.

3.2.1. Local Strategy

In this strategy the threads take a query from Q_{PQ} (*Private Queue of Queries*), which is the queue where the queries arrive from the broker, and from that query is generated an initial task (with the fields described in the previous Section 3.2), and it is added to Q_{PT} (*Private Queue of Tasks*). As we described in the previous section, if it is necessary, when a task is processed it will generate more tasks, which are also added in the same Q_{PT}. Remember that Q_{PQ} and Q_{PT} are local and private to the thread. The queue Q_{SM} (*Shared Queue of Messages*) is not used in this strategy.

The above implies that each thread processes a query completely and isolated from the rest, i.e. each thread is able to process a query with no communication with others threads, avoiding synchronization instructions and message passing.

Taking account (as we described in Section 3.2) that the index is global and shared for all the threads, and also the queries are distributed in a circular manner, the Algorithm 6 describes the search process of this strategy. In line 4, the initial task of a query is stored in the queue Q_{PT}. Then, a task of the queue is extracted (line 6) and processed (line 7). The function that process a task is called `executeTask(t)`, which changes depending on the index we use, because each index define its own parameters for its tasks (Section 3.2). This function returns a list with the new tasks generated by the task parameter `t`. Finally, the new generated tasks are inserted in Q_{PT} (line 8) to be processed in the next cycle of the while of line 1.

Algorithm 6 Search for *local strategy*.

{*executeTask*(t): Processes the task t and returns the list of new generated tasks.}

ThreadQueryProcessing()

```
 1: while true do
 2:     if empty(Q_PT) then
 3:         initial_task ← nextQuery(Q_PQ)
 4:         Q_PT.insert(initial_task)
 5:     end if
 6:     task ← nextTask(Q_PT)
 7:     taskList ← executeTask(task)
 8:     for all t ∈ taskList do
 9:         Q_PT.insert(t)
10:     end for
11: end while
```

3.2.2. Bulk-Circular Strategy

This strategy processes queries using the BSP (Bulk Synchronous Parallel) model [57], which sets a synchronous behavior for the threads involved. To implement the BSP model and create a sequence of steps (*Supersteps* in the BSP model), we used the OpenMP directive `#pragma omp barrier`.

In this strategy each thread uses the three queues Q_{PQ}, Q_{PT} and Q_{SM}. Each thread stores in its own Q_{SM} the messages addressed for other threads. Q_{SM} is a globally shared queue by means of the `shared()` option of the directive `#pragma omp parallel` of OpenMP, thus all the threads can access to the Q_{SM} of the rest. A message is a task plus the ID of the target thread.

The above implies that the generated tasks are not just stored in the Q_{PT}, as it is in the *Local* strategy, but also the tasks addressed for other threads (as a message) are stored in the Q_{SM}. Each thread choose the target for a message in a circular manner.

The Algorithm 7 describes the search process for this strategy, which is composed by two *supersteps*. The first one, initialize the queue Q_{SM} deleting all its elements, then tasks are taken from Q_{PT} (line 3) and they are processed until reach R distance evaluations (line 2). The function `executeTask(t)` of line 8 is in charge of processing tasks, therefore into this function the distance evaluations are performed, and it increase the value of the variable *limit* (used in line 2). We set the target thread of the new generated tasks following a circular distribution, and if the target is the own thread, then the task is stored in the Q_{PT} (line 11), or in the other case, the task is stored in the corresponding Q_{SM} (line 13). If Q_{PT} is empty, then a new query is taken from Q_{PQ}, and the initial tasks is added to Q_{PT}. When R distance evaluations are reached, the second superstep begins, where each thread reads the Q_{SM} of the rest (line 18) and extract the messages addressed for him. All these messages are inserted as a task in the Q_{PT} (line 22), and then the first step begins again, and so on. It is noteworthy that read and write to queues Q_{SM} are made in different supersteps, which implies no concurrency conflict. The number R determines how long is the processing of the first superstep, therefore it has an impact on the balance between both supersteps. We empirically found the value of R.

3.2.3. Bulk-Critical Strategy

This strategy is very similar to *Bulk-Circular* strategy (Section 3.2.2), but uses *crtitical regions* of OpenMP (Section 2.4) to implement message passing. A critical region is a region of code executed for just one thread at a time, i.e. if a thread try to execute a sentence of a critical region that is already in use for other thread, will have to wait until the region be available.

Algorithm 7 Search in *Bulk-Circular* strategy.

{tid: ID of the thread.}
{P: Total quantity of threads.}
{$executeTask(t)$: Processes the task t and returns a list with the new generated tasks.}

ThreadQueryProcessing(tid)

```
 1: while true do
 2:     while limit < R do
 3:         if Q_PT.empty() == true then
 4:             initial_task ← nextQuery(Q_PQ)
 5:             Q_PT.insert(initial_task)
 6:         end if
 7:         task ← nextTask(Q_PT)
 8:         taskList ← executeTask(task)
 9:         for all t ∈ taskList do
10:             if t.targetThread == tid then
11:                 Q_PT.insert(t)
12:             else
13:                 Q_SM[tid].insert(t)
14:             end if
15:         end for
16:     end while
17:     #pragma omp barrier
18:     for i = 0; i < P; i++ do
19:         if i != tid then
20:             for j = 0; j < Q_SM[i].size(); j++ do
21:                 if Q_SM[i].getTask(j).targetThread == tid then
22:                     Q_PT.insert(Q_SM[i].getTask(j))
23:                 end if
24:             end for
25:         end if
26:     end for
27:     #pragma omp barrier
28:     Q_SM[tid].clear()
29: end while
```

The fact of using critical regions has the advantage of avoiding synchronization instructions. But, it has the disadvantage that the sequential access to the critical regions can be a bottleneck.

The Algorithm 8 shows the implementation of the search process for this strategy, using critical regions with the directive `#pragma omp critical`. As in the

previous strategy, the target of a message is determined by a circular distribution.

Algorithm 8 Search process of *Bulk-Critical* strategy.

{tid: ID of the thread.}
{P: Total quantity of threads.}
{$executeTask(tk)$: Processes the task tk and returns a list with the new generated tasks.}

ThreadQueryProcessing(tid)

```
 1: while true do
 2:     while limit < R do
 3:         if Q_PT.empty() == true then
 4:             initial_task ← nextQuery(Q_PQ);
 5:             Q_PT.insert(initial_task);
 6:         end if
 7:         task ← nextTask(Q_PT);
 8:         taskList ← executeTask(task);
 9:         for all t ∈ taskList do
10:             if t.targetThread == tid then
11:                 Q_PT.insert(t);
12:             else
13:                 #pragma omp critical (exchange){
14:                 Q_SM[tid].insert(t);
15:                 }
16:             end if
17:         end for
18:     end while
19:     #pragma omp critical (exchange){
20:     for i = 0; i < P; i++ do
21:         if i != tid then
22:             for j = 0; j < Q_SM[i].size(); j++ do
23:                 if Q_SM[i].getTask(j).targetThread == tid then
24:                     Q_PT.insert(Q_SM[i].getTask(j));
25:                     erase(Q_SM[i].getTask(j));
26:                 end if
27:             end for
28:         end if
29:     end for
30:     }
31: end while
```

Table 3.1: General Features

Processor	2xIntel Quad-Xeon (2.66 GHz)
L1 Cache	8x32KB + 8x32KB (inst.+data) 8-way associative, 64byte per line
L2 Unified Cache	4x4MB (4MB shared per 2 procs) 16-way associative, 64 byte per line
Memory	16GBytes (4x4GB) 667MHz DIMM memory 1333 MHz system bus
Operating System	GNU Debian System Linux kernel 2.6.22-SMP for 64 bits

3.2.4. Bulk-Local Strategy

This is similar to *Local* strategy (Section 3.2.2), but the tasks are synchronously processed in a superstep. This means that after each thread has performed R distance evaluations, a synchronization barrier is applied, and the process is repeated with the next R distance evaluations, and so on. Thus, there is no Q_{SM} in this strategy.

This is a local strategy, where each thread processes a query completely, but performs batch processing synchronously. The reason for implementing this strategy is that the synchronous processing under high query traffic has shown very good results in previous papers ([38, 34]).

3.3. Experimental Results

All the experiments were executed in a node with two CPUs Intel Quad-Xeon, each one with four cores. The general features are shown in the Table 3.1.

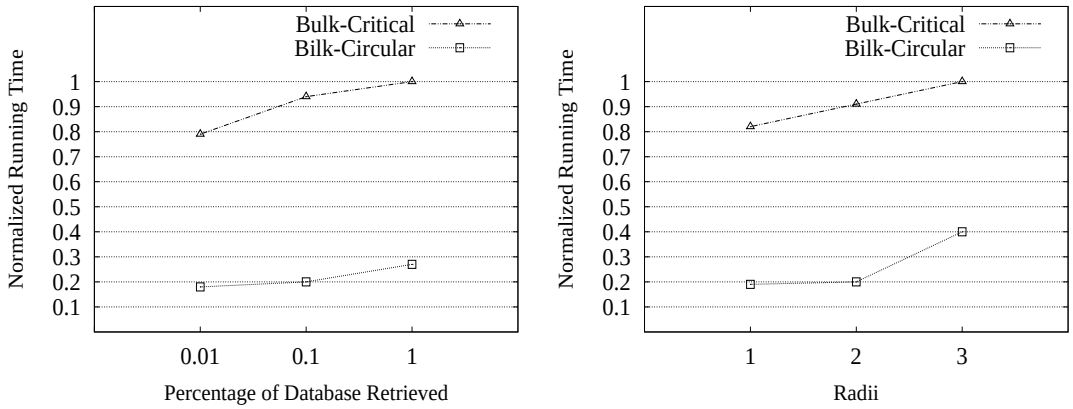
The experiments were performed with two databases:

- ***Words*** **:** Spanish dictionary with 51589 words. The *edition distance* (or *Levenshtein distance*) [32] was used with radius 1, 2 and 3 because these radii were also used in previous works ([41, 35, 40]). This distance function returns the minimum quantity of insertion, deletion or replaces for one word becomes another. The query log for this database was a file with 40,000 queries, taken from the Chilean domain todo.cl.

- ***Images*** **:** This database was made from a collection of 40,701 vectors that represent images from the NASA (available in the Metric Space Library of sisap.org), which were used as a probability distribution to generate random vectors until reach 120,000 images of dimension 20. The *euclidean distance* was used as distance function. We used radii that retrieve 0.01%, 0.1% and 1% of the database per query (on average). These values were used in previous works [16, 41, 40]. The 80% of the database was used for the construction of the index an the 20% left used as the query file.

The experiments were normalized to the highest value of the experiment (this is to distinguish better the differences between strategies). We used situations of low and high query traffic.

The Figure 3.3 shows the running time for *Bulk-Circular* and *Bulk-Critical* strategies. The experiment was made using a high query traffic situation and using the *EGNAT* index, but a similar behavior was observed with the other indexes. The *Bulk-Critical* strategy shows a very high running time, mainly due to the high access to the critical regions, and as is known ([13]), it considerably decrease the performance of the program. Because of this, this strategy was discarded for the forward experiments.



(a) DB Images (b) DB Words

Figure 3.3: Running time for strategies *Bulk-Circular* and *Bulk-Critical*, using the *EGNAT* index with a high query traffic.

The Figure 3.4 shows the experiments using all the strategies with all the indexes under a high query traffic situation, for the databases *Words* and *Images*. The Figure 3.5 shows the same experiments, but under a low query traffic situation. These latter two figures show normalized running time over the highest value of the experiment using the corresponding index, therefore it is noteworthy that we cannot compare the performance between different indexes in these figures.

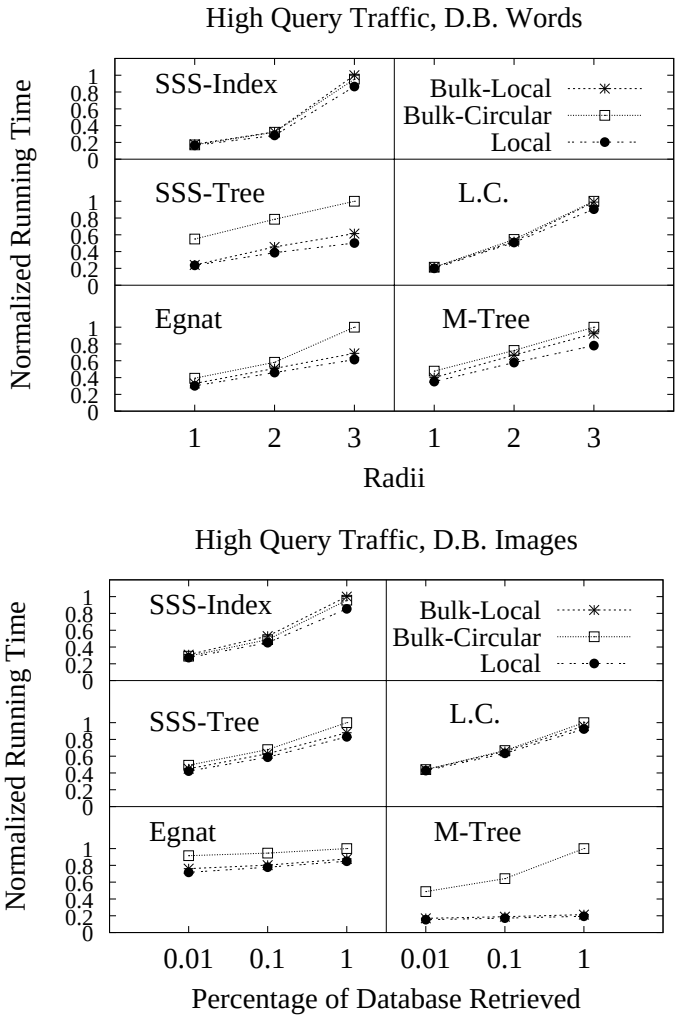


Figure 3.4: Running time under a **high** query traffic.

We can observe that under a high query traffic, the *Local* strategy gets a better performance for all the indexes. This is mainly because the high cost of the synchronization in the *Bulk-Circular* strategy. With regard to the *Words* database, the *SSS-Index* is able to discard few elements with a high radius, which is a behavior seen before ([9]). The index with less variation among the strategies for both databases was the *LC*, and it is because the size of a task involves a complete cluster, which

allows a good balance between the processing and communication steps in the *Bulk-Circular* strategy under a high query traffic.

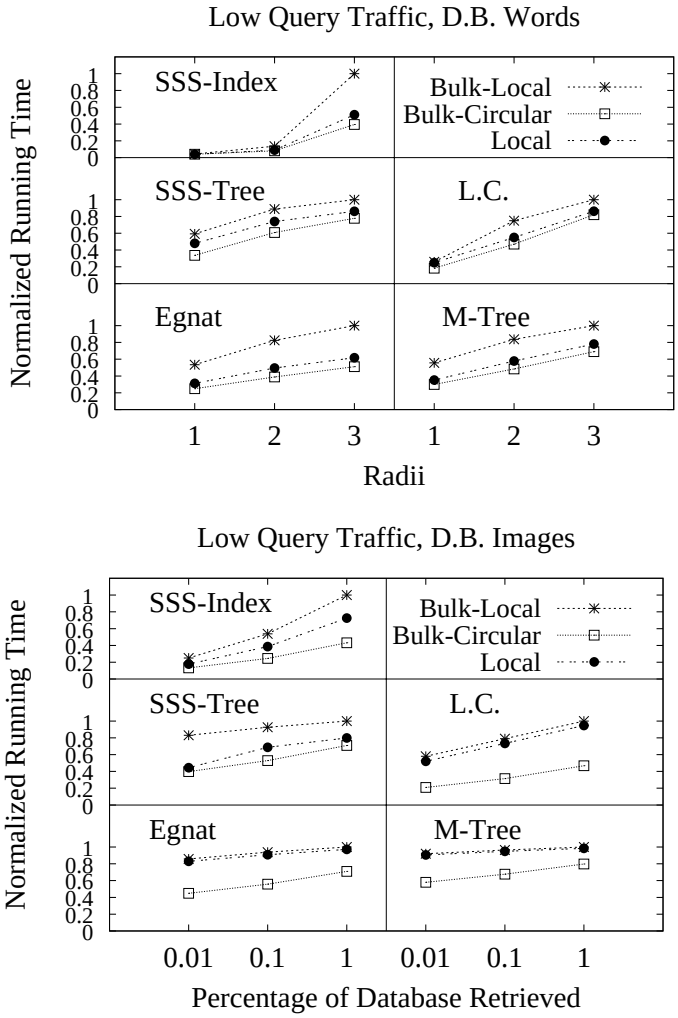


Figure 3.5: Running time under a **low** query traffic.

Under a low query traffic (Figure 3.5), we observe that the *Bulk-Circular* strategy takes advantage. The reason is mainly because this strategy reduces the idle time of the threads that are waiting for the next query. This strategy distributes the

tasks of each query among all the threads, thus it makes a better use of the idle time, and under a low query traffic this feature gives advantage.

Favorable Arrival

Time	Local				Bulk–Circular			
	T1	**T2**	**T3**	**T4**	**T1**	**T2**	**T3**	**T4**
1	q1	q2			q1	q1	q1	q1
2	q1	q2	q3		q2	q2	q2	
3	q1	q2	q3	q4	q3		q3	q4
4	q1	q5	q6		**q5**	**q5**	**q5**	**q5**
5		q5	q6	q7	q6	q6	q6	q7
6	q8	q5	q6	q7	q7			
7	q9	q5	q10		q8	q10	q10	q10
8	q9		q10	q11	q9	q9	q9	q9
9	q9	q12	q10	q11	q11	q12		q11
10	q9							
11								
12								

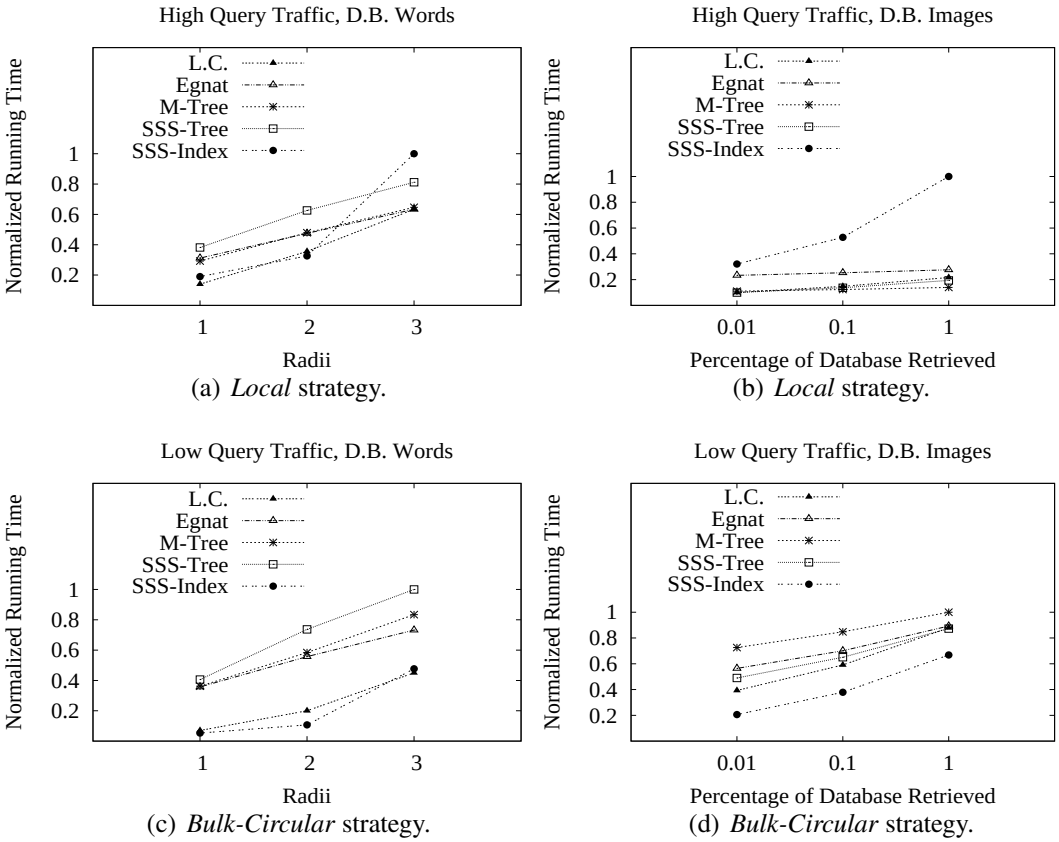
Unfavorable Arrival

Time	Local				Bulk–Circular			
	T1	**T2**	**T3**	**T4**	**T1**	**T2**	**T3**	**T4**
1	q1	q2			q1	q1	q1	q1
2	q1	q2	q3		q2	q2	q2	
3	q1	q2	q3	q4	q3		q3	q4
4	q1	q6			q5	q5	q5	q5
5	q5	q6	q7		q6	q6	q6	
6	q5	q6	q7	q8	q7		q7	q8
7	q5	q10			q9	q9	q9	q9
8	q5	q10	q11		q10	q10	q10	
9	**q9**	q10	q11	q12	q11		q11	q12
10	**q9**							
11	**q9**							
12	**q9**							

Figure 3.6: Cases of favorable and unfavorable arrival of queries and the distribution of their tasks, under a low query traffic. Blank spaces are idle time of the thread

To illustrate better the latter point, the Figure 3.6 shows an example of two different distributions of queries (under a low query traffic), where queries that require different quantity of distance evaluations arrive to the system. The rate of arrival time of queries is as follows: in the first unit time 2 queries arrive, in the second one 1 query, in the third one 1 query, and then this pattern is repeated. Each instance of q_i that appears in the picture represents a task to be solved, thus the query $q1$ in the figure requires 4 tasks to be solved. The first distribution (on the left), stands for a favorable arrival of queries, i.e. the queries arrive to the system in such a way that allow an optimal distribution of the tasks of each query, thus all the threads perform (approximately) the same quantity of distance evaluations. The second (on the right), is an unfavorable distribution, where the first thread processes all the queries that require more quantity of distance evaluations to be solved. In

both (extreme) cases of low query traffic, the *Bulk-Circular* strategy gets a better throughput (solved queries per unit time).



(a) *Local* strategy. (b) *Local* strategy.

(c) *Bulk-Circular* strategy. (d) *Bulk-Circular* strategy.

Figure 3.7: Comparison of the indexes with the *Local* strategy for high query traffic and *Bulk-Circular* for low query traffic. Values normalized over the highest value observed in the corresponding experiment.

The Figure 3.7 shows the comparison between the indexes, using the *Local* strategy for high query traffic, and *Bulk-Circular* for low query traffic. This figure has the objective of observing which index adapts better to the different strategies. An index with a good performance taking account both traffic situations was the *List of Cluster* (*LC*). One of the reasons for this behavior is that a task of this index

requires a fixed quantity of distance evaluations, because a task covers a complete cluster, and this improves the load balancing.

The Figure 3.8 compares the speed-up of each index over its sequential counterpart, using the database *Words* with radius 3. The highest point of speed-up is reached with *Bulk-Circular* under low query traffic. This result is supported by the Figure 3.9, which shows the efficiency $E = (\sum(w_i/max_w))/P$ (using the *LC*) of the strategies *Bulk-Circular* and *Local*, where w_i is the workload of the thread i; max_w is the maximum workload that some thread has made, and P is the quantity of threads. This experiment shows that the idleness of the threads is higher when the *Local* strategy is used under a low query traffic.

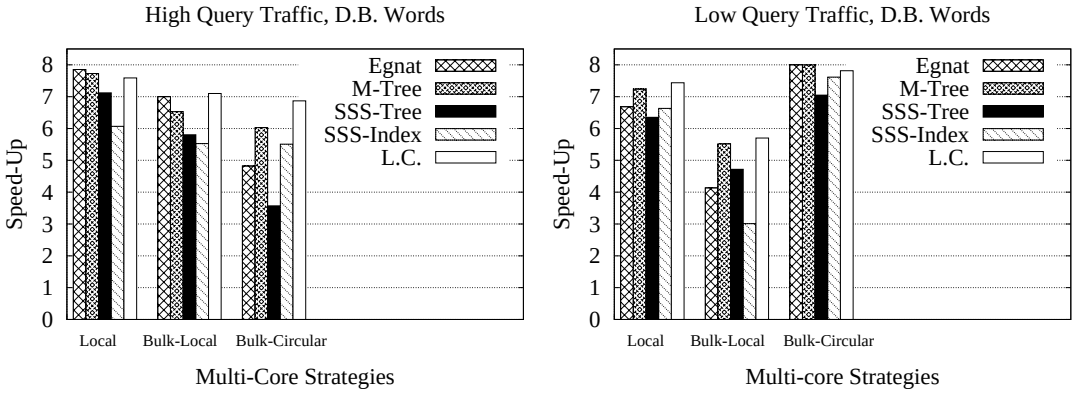


Figure 3.8: Speed-up of the indexes over its sequential counterpart, using the *Words* database with radius 3.

3.3.1. Hybrid Strategy

This strategy is able to apply an exchange between the *Local* and *Bulk-Circular* strategies depending on the current query traffic. When the number of the waiting queries C_p satisfies $C_p > P * C_{max}$ (P: quantity of threads, C_{max}: threshold number of

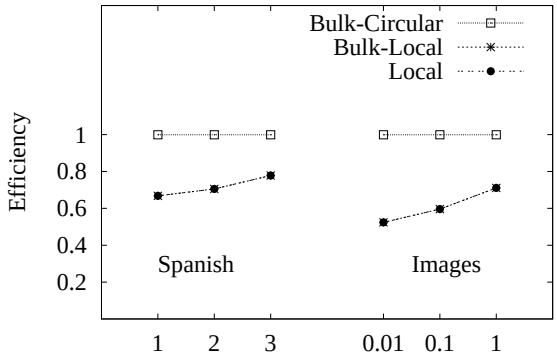


Figure 3.9: Efficiency (average time of threads processing tasks) of the multi-core strategies using the *LC* under a low query traffic.

queries that separates a low and high query traffic situation), the tasks are processed using the *Local* strategy and when the number of waiting queries is low enough is applied the *Bulk-Circular* strategy. It is noteworthy that the change of strategy is applied to all the threads at the same time.

The Figure 3.10 shows the throughput (solved queries per unit time) for the *Local*, *Bulk-Circular* and *Hybrid* strategies using the *LC*. The *Hybrid* strategy shows the highest throughput, because it exploits the advantages of both strategies.

3.3.2. Local Distribution of the database

As we explained in Section 3.2, all the experiments so far has been performed with a global shared database, i.e. there is just one index in memory, and all the threads access to it to process the queries.

This section proposes to distribute the index in P partitions (P=quantity of threads), and assign each partition to a different thread. Thus, each thread will process its queries accessing to its own index with local data, which is just a part

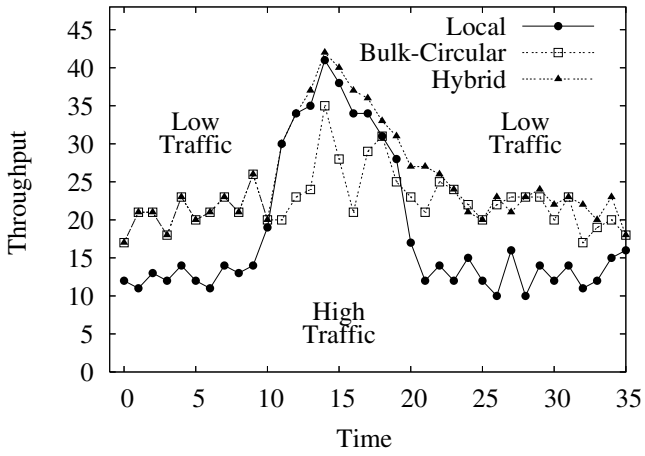


Figure 3.10: Throughput: queries completely solved per unit time, using the *LC* index.

of the complete database. To do this, the elements of the databases *Words* and *Images* were distributed in a circular manner among the threads, and then each thread creates its own index with its assigned elements.

This distribution has the disadvantage that each thread has an index that represents just a portion of the whole database, therefore, all the queries must be processed by all the threads. But, has the advantage that each thread processes the queries on a index with less elements. To avoid synchronizations, each thread writes its results in a different memory address (dynamically allocated), and the final result is the union of the partial results of each thread.

The Figures 3.11 and 3.12 show the running time and quantity of distance evaluations respectively, normalized to the highest value of the experiment, using the databases *Words* and *Images* under a high query traffic. The proposed distribution in this section, where each thread creates its own local index was called *Distributed DB* in the figures, and the distribution used so far, where all the threads use just one

shared global index was called *Global DB*. Both distributions use *Local* strategy (Section 3.2.1), therefore each thread solve its queries with no communication with the rest of threads, and no synchronizations, in both distributions. In the case of *Distributed DB*, the broker must send each query to all the threads, and for *Global DB* each query is sent to just one thread.

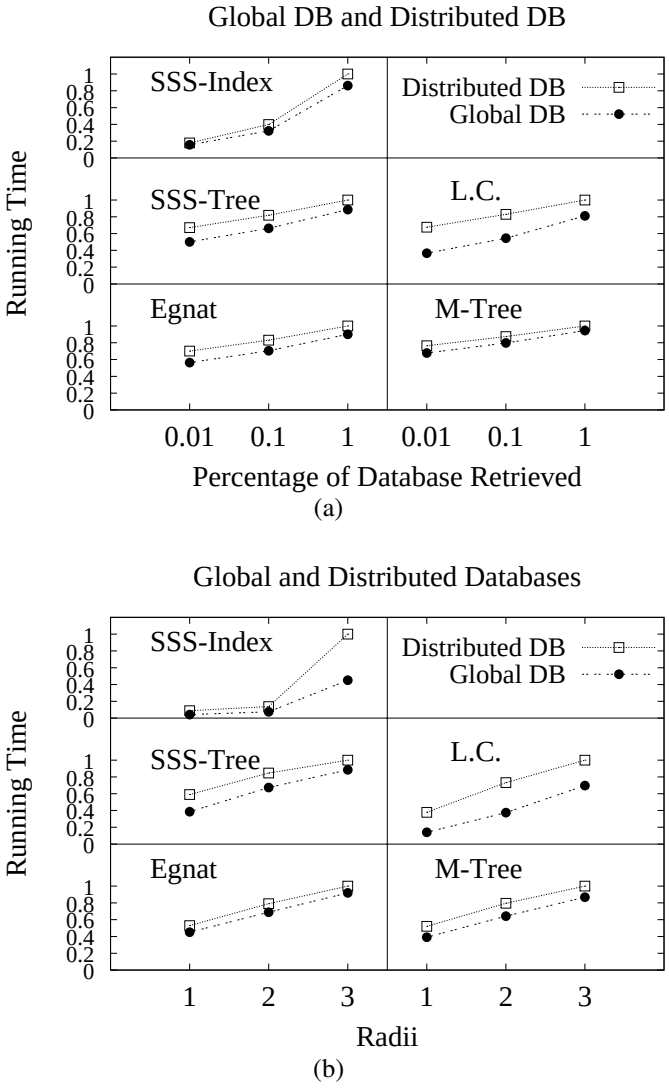


Figure 3.11: Normalized running time using the databases **a)***Images* and **b)***Words*.

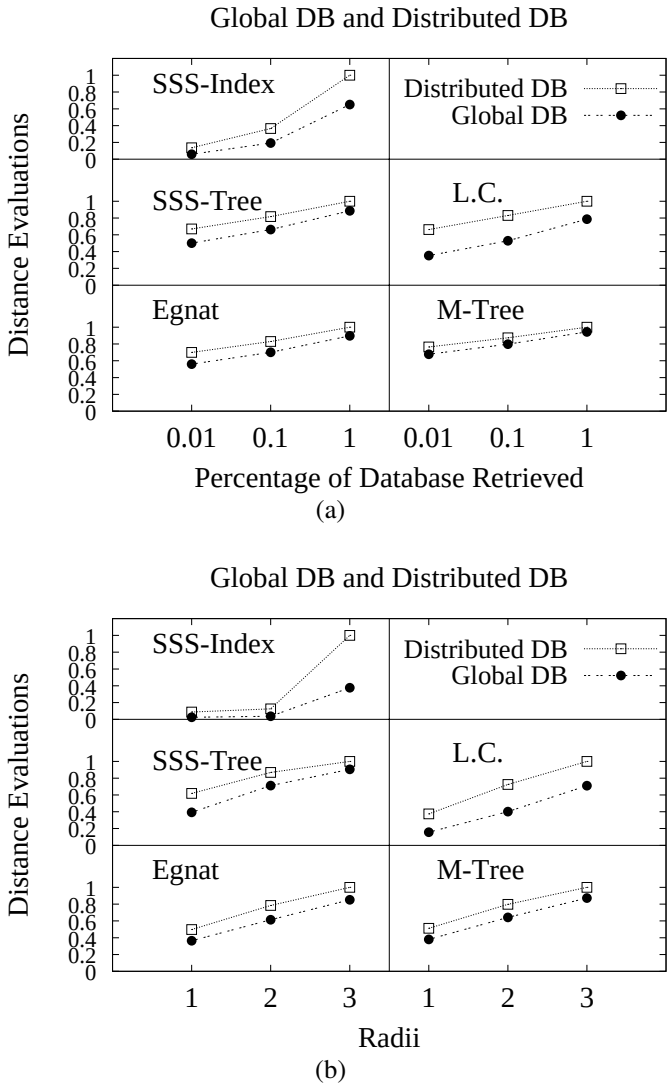


Figure 3.12: Normalized average of distances evaluations per query, using the databases **a)***Images* and **b)***Words*.

Figure 3.12 shows that when the elements of the database are distributed to create local indexes (*Distributed DB*), the number of distance evaluations is increased. This is mainly because the selected centers (or pivots) in the *Global DB* method are of higher quality [38] and more representative of the database. This allows a better

discard and pruning of elements, due to the global centers (or pivots) are selected taking account all the elements of the database. The results regarding running time (Figure 3.11), where the *Global DB* method take advantage was an expected behavior given the number of distance evaluations (observed in Figure 3.12), and the high cost of a distance evaluation between two elements.

3.4. Conclusions

In this chapter several algorithms have been proposed on a multi-core platform to process queries in metric spaces. A set of representative and very used indexes in the technique literature were used to show the generality of our proposed algorithms. These algorithms implement asynchronous multi-thread processing (*Local* strategy) and bulk-synchronous processing (*Bulk-Circular*, *Bulk-Local* and *Bulk-Critical* strategies), where the latter is an implementation of the BSP model.

The *Bulk-Critical* strategy shows the lowest performance, mainly due to the high number of accesses to critical regions (of OpenMP), and the high cost each access implies. The code of a critical region is sequentially executed by just one thread at a time, thus its high access decrease the performance.

The *Local* strategy shows the highest performance under a high query traffic situation, achieving up to 7.8x of speed-up. In this strategy each thread process its queries completely, without communication with others threads, and avoiding synchronizations between them.

The *Bulk-Circular* strategy shows the highest performance under a low query traffic situation (up to 7.9x of speed-up). In this strategy each thread distributes the *tasks* of its queries among all the other threads, which means that all the threads

cooperate to solve every query. This takes advantage under a low query traffic mainly because reduces the idle time of the threads.

According to the previous results, we propose a *hybrid* strategy, which is able to change between the *Local* and *Bulk-Circular* strategies, depending on the current query traffic. This strategy shows the best performance in a scenario where the traffic can change, because it is able to exploit the advantages of both, *Local* and *Bulk-Circular* strategies.

In Section 3.3.2 we compared two distributions of the database. The first, distribute the elements of the database among the threads, and each thread creates its own index. The second, keep just one global index in memory. The latter shows the best performance regarding to the quantity of distance evaluations and running time. This is due to the quality that show the global centers (or pivots), increasing the percentage of discarded elements.

Chapter 4

Distribution and Search Strategies on a single-GPU

In the current technological context, one of the most promising alternatives for the acceleration in search operations is the use of Graphics Processing Units (GPUs). The GPU presents several levels of parallelism to explore, given that it is able to execute in parallel a high quantity of threads organized in CUDA Blocks and grids. Also, it is compatible with a multi-core program, i.e. it is possible that a multi-core program be in execution and in parallel each thread manage a different GPU. As we showed in Section 2.5, these architectures have a complex memory hierarchy which include a low latency shared memory, texture memory, constant memory, global memory, and read-only cache memories to speed up access to texture and constant memories. Empirical studies show that it is crucial to efficiently exploit this memory system to achieve a significant performance improvement when using GPUs to accelerate a given application [48].

In this chapter we propose and compare similarity search algorithms using metric indexes on a single GPU based on the programming model CUDA [22].

The programming on GPU could be very tricky, mainly because it is very easy to add divergence in the sequence of instructions of threads of the same warp. As explained in Section 2.5, when threads of the same warp execute different instructions, those instructiones are sequentially executed, decreasing the performance. The factors that add divergence are several, such as: (1) different execution paths due to conditional sentences, (2) no contiguous access to device memory, (3) access to the same bank of data of shared memory.

Taking account all the features of the GPU mentioned above, we decided to implement and map on GPU the *List of Clusters* (*LC*) [16] and *SSS-Index* [8] indexes since (1) their good results shown in the previous Chapter 3, (2) they are two of the most popular non-tree structures that are able to prune the search space efficiently and (3) they hold their indexes on dense matrices and exhibit certain regularity in the access pattern, which are suitable features for mapping algorithms onto GPUs. We are not affirming that these indexes are the only suitable for the GPU, but their properties make them good candidates to be it. Besides, finding the best metric index for GPU is not an objective of this thesis; we mainly want to show the great performance achievable using a metric index on GPU compared to traditional sequential and multi-core approaches.

Due to the complexity and restrictions of the GPU, we found different problems for both kinds of queries, *range* and *k*NN queries, thus we applied different parallelization strategies for each type. We also, included an exhaustive search for both kinds of query, which were used as base for comparison.

In all our proposed algorithms, each CUDA Block (block of threads executed

in GPU) is in charge to solve one query completely, which implies that the kernel that process queries will have as many CUDA Blocks as queries to be solved. The fact of solving a query with just one CUDA Block allows to use synchronization, which is available just for threads of the same CUDA Block. The synchronization is required to solve range and *k*NN queries using the *LC* and *SSS-Index*, and the alternative method to synchronize threads is to launch a new kernel, but that is more costly than use the synchronization function available for threads of the same CUDA Block. Therefore, we are exploiting two levels of parallelism: *coarse* and *fine grained parallelism*. We exploit coarse grained parallelism when we process a set of Q queries in parallel in just one launch of kernel. The fine grained parallelism is exploited when we process each query with a set of threads (all the threads of the CUDA Block).

Because the above, each kernel is launched with Q CUDA Blocks (Q=quantity of queries to be processed), optimizing the number of threads per CUDA Block. The maximum quantity of CUDA Blocks in one kernel (for our model of GPU) is 65535^3. If Q exceeds the maximum allowed quantity of CUDA Blocks, then consecutive kernels must be launched. But, in the experiments of this chapter it was necessary just one launch of kernel.

The query to be processed by the CUDA Block is always stored in *shared memory*, which is (as described in Section 2.5) a reduced size memory allocated inside each multiprocessor, therefore it has a very low latency. The data stored in this memory are shared only by the threads of the same CUDA Block.

Regarding the management of results for a range query, all our proposed strategies on GPUs manage it in the same following way. Despite that we could require and retrieve different information from each element, for the purposes of this thesis,

we increase a counter for each found result, to track the number of elements found by each thread, but no further processing or transfers are performed.

The organization of this chapter is as follows. In Section 4.1 we show our proposals to solve range queries and in Subsection 4.1.4 we show the experimental results using a single GPU. In Section 4.2 we describe our proposals to solve *k*NN queries using exhaustive and indexing searches, and in Subsection 4.2.4 we show the experiments processing *k*NN queries. Finally in Section 4.3 we summarize the main conclusions of the chapter.

4.1. Processing Range Queries on a single-GPU

In this section we describe the mapping of three range search algorithms onto CUDA-enabled GPUs: a brute-force approach and two index-based search methods (*LC* and *SSS-Index*).

In Section 4.1.1, we describe our brute force algorithm to solve range queries on GPU, which does not use any method to discard objects, thus it calculate the distances between all the elements of the database against the query. In Sections 4.1.2 and 4.1.3 we describe the mapping of the *LC* and *SSS-Index* respectively. In both indexes we used algorithms composed by steps delimited by synchronizations functions. We used in both cases three steps: (1) To copy the query to shared memory, (2) To discard elements using the index, and (3) To compare the non-discarded elements against the query.

4.1.1. Brute Force Algorithm on a single-GPU Processing *Range* Queries

A brute force algorithm is a general problem-solving technique that consists of checking each possible solution and see if it satisfies the statement of the problem. In the particular case covered in this section, each CUDA Block processes a different query and within a CUDA Block, each thread computes the distance between the query and a subset of the elements of the database, avoiding to use an intermediate structure or index.

The Algorithm 9 shows our brute force technique, where the *DB* matrix represents the database of size $D \times SIZE_{DB}$, D is the dimension of its elements[1] and $SIZE_{DB}$ is the size of the database, which has been uploaded previously to *device memory*. Queries are also uploaded into device memory and the threads of each CUDA Block cooperate to transfer their associated query to the *shared memory* to accelerate its access. The latter is the first step of the Algorithm 9 (line 4) in the `range_search_BF()` kernel. Afterwards, threads compute the distance between the query and the elements of the database following a circular distribution (line 10). Most work is performed within the device distance function. Database elements are stored column-wise to increase the chances of coalesce memory accesses when computing these distances, since that way consecutive threads have to access to adjacent memory locations.

As we mentioned before (Chapter 2), in sequential computing the brute force algorithm has been widely outperformed by indexing based algorithms. In the following sections we present our index based proposals for a single GPU, using

[1] In the case of the words database, D is the maximum length of a word.

Algorithm 9 Brute force search kernel to process range queries on GPU.

```
{tid is the ID of the thread inside the CUDA Block}
{Each column of DB is an element of the database}
{SIZE_DB is the number of elements of the database}
{D is the dimension of the elements of the database}
{T_Block is the number of threads per CUDA Block}

range_search_BF(float **DB, float *Query, float range)
 1: __shared__ float query[D];
 2:
 3: for (i = tid; i < D; i+=T_Block) do
 4:     query[i] = Query[i]
 5: end for
 6:
 7: __syncthreads()
 8:
 9: for (j = tid; j < SIZE_DB; j+=T_Block) do
10:     if distance(DB, j, query) <= range then
11:         found()
12:     end if
13: end for

distance(float **M1, int col, float *M2)
   dist = 0
   for (i = 0; i < D; i++) do
      dist += (M1[i][col] − M2[i]) ∗ (M1[i][col] − M2[i])
   end for
   dist = sqrtf(dist)
   return dist
```

the indexes *LC* and *SSS-Index*.

4.1.2. List of Cluster (LC) on a single GPU processing *range* queries

As we mentioned in Section 2.2.5, the *LC* is an index based on covering radius (Section 2.2). It is composed by balls, and each ball contains: (1) a fixed number of elements, (2) an object as center of the ball, and (3) the covering radius, which is the distance from the center to the farthest element of the ball. To implement the *LC*

on GPU, we represented its data structure with 3 matrices denoted as *CENTERS*, *RC* and *CLUSTERS* in Algorithm 10. *CENTERS* is a $D \times SIZE_{cen}$ matrix (D is the dimension of the elements and $SIZE_{cen}$ is the number of centers), where each column represents the center of a cluster, *RC* is an array that stores the covering radius of each cluster, and *CLUSTERS* is a $D \times SIZE_{clu}$ matrix ($SIZE_{clu}$ is the number of elements in all the clusters) that holds the elements of each cluster. We stored column-wise the elements in the matrices to favor coalesce memory accesses. The latter is reflected when consecutive threads access to consecutive elements or centers of the index, because as we mentioned in Section 2.5, when threads of the same warp access to consecutive elements, the read/write operations are coalesced.

Algorithm 10 shows the pseudocode of the main CUDA kernel that solves one range query (*query*, *range*) using one CUDA Block with the *LC* index. The index is offline created, thus the search algorithm starts with the whole index previously loaded in the device memory of the GPU. Each CUDA Block processes a different query, which is transferred from device memory to shared memory (line 6) since the query is accessed by all the threads of the CUDA Block when performing distance evaluations. Once the query has been saved into the shared memory, the *for* loop starting at line 12 iterates over the different centers of clusters. Each thread computes the distance between the query and a subset of elements of *CENTERS* following a circular distribution. Most work is performed again within the device function `distance()`. If the distances are lower than *range*, the respective centers are appended to the list of results in `found()` (line 15). Clusters are marked for exhaustive search at line 17 only if their respective cluster balls have some intersection with the query ball. A property of this index (given by its construction) is that the exhaustive search over a cluster can be pruned if the query ball is totally contained

in a given cluster ball. If this is the case (line 18), then we do not consider the subsequent clusters and delimit the number of clusters at line 19.

Finally, the *for* loop starting at line 26 processes all the elements of the selected clusters as in the Brute Force technique.

4.1.3. SSS-Index on a single-GPU Processing *Range* Queries

As we showed in Section 2.2.3, the *SSS-Index* is an index based on pivots (Section 2.2). It is composed by: (1) a table of distances between key elements (called pivots) of the database and the each remaining element, and (2) the elements themselves. The pivots are selected using the *SSS* algorithm (Section 2.2.3), and the table of distances is built in a offline way, previous to the search. To be able of mapping the *SSS-Index* on the GPU, we represented its data structure with 3 matrices denoted as *PIVOTS*, *DISTANCES* and *DB* in the Algorithm 11. *PIVOTS* is a $D \times SIZE_{Piv}$ matrix (D is the dimension of the elements and $SIZE_{Piv}$ is the number of pivots) where each column represents a pivot, *DISTANCES* is a $SIZE_{Piv} \times SIZE_{DB}$ matrix ($SIZE_{DB}$ = number of elements of the database) where each element is the distance between a pivot and an element of the database, and *DB* is a $D \times SIZE_{DB}$ matrix where each column represents an element of the database. As we did in the previous section, the index information is stored column-wise to favor coalesce memory accesses.

Algorithm 11 shows the pseudocode of the main CUDA kernel that solves one range query (*query*, *range*). First, each CUDA Block transfers its associated query to shared memory due to its frequent access (line 5). Once a synchronization (line 8) ensures the query has been copied before being accessed, each thread performs the

Algorithm 10 Range search kernel to process range queries on GPU using the *LC*.

{*tid* is the ID of the thread inside the CUDA Block}
{*D* is the dimension of the elements of the database.}
{B_{Size} is the number of elements of each cluster.}
{$SIZE_{clu}$ is the total number of elements in the clusters.}
{$SIZE_{cen}$ is the number of centers of clusters.}
{T_{Block} is the number of threads per CUDA Block.}

range_search_LC(float **CENTERS*, float **RC*, float **CLUSTERS*, float **Query*, float *range*)

```
 1: __shared__ float query[D]
 2: __shared__ int exhaustive[SIZE_cen]
 3: __shared__ int minC=SIZE_cen
 4:
 5: for (i = tid; i < D; i+=T_Block) do
 6:     query[i] = Query[i]
 7: end for
 8:
 9: __syncthreads()
10:
11: {Each thread tries to discard a different cluster}
12: for (j = tid; j <minC; j+=T_Block) do
13:     dist = distance(CENTERS, j, query)
14:     if dist <= range then
15:         found()
16:     end if
17:     exhaustive[j] = dist <= RC[j] + range
18:     if dist < RC[j] - range then
19:         atomicMin(&minC, j)
20:     end if
21: end for
22:
23: __syncthreads()
24:
25: {The distances between each non-discarded cluster and the query are calculated}
26: for (j = tid; j < SIZE_clu && j/B_Size <= minC; j+=T_Block) do
27:     if exhaustive[j/B_Size] == 1 then
28:         if distance(CLUSTERS, j, query) <= range then
29:             found()
30:         end if
31:     end if
32: end for
```

distance evaluations between the query and a subset of pivots following a circular distribution (line 11). And finally, the rows of *DISTANCES* are distributed across

threads (line 18) to test if their respective elements of the database can be discarded (line 21). For every non discarded element, a distance evaluation is performed (line 27).

Algorithm 11 Range search kernel to process range queries on GPU using the *SSS-Index*.

{tid is the ID of the thread inside the CUDA Block}
{D is the dimension of the elements of the database.}
{$SIZE_{DB}$ is the number of elements of the database.}
{$SIZE_{Piv}$ is the number of pivots.}
{T_{Block} is the number of threads per CUDA Block.}

range_search_SSS(float **PIVOTS*, float ***DISTANCES*, float ***DB*, float **Query*, float *range*)

```
 1: __shared__ float query[D]
 2: __shared__ float dist_piv[SIZE_Piv]
 3:
 4: for (i = tid; i < D; i+=T_Block) do
 5:     query[i] = Query[i]
 6: end for
 7:
 8: __syncthreads()
 9:
10: {The distances between each pivot and the query are calculated}
11: for (j = tid; j <SIZE_Piv; j+=T_Block) do
12:     dist_piv[j] = distance(PIVOTS, j, query)
13: end for
14:
15: __syncthreads()
16:
17: {Each thread tries to discard a subset of element with all the pivots}
18: for (j = tid; j <SIZE_DB; j+=T_Block) do
19:     discarded = 0
20:     for (i = 0; i < SIZE_Piv; i++) do
21:         if dist_piv[i] < DISTANCES[i][j] - range || dist_piv[i] > DISTANCES[i][j] + range then
22:             discarded = 1
23:             break
24:         end if
25:     end for
26:     if discarded == 0 then
27:         if distance(DB, j, query) <= range then
28:             found()
29:         end if
30:     end if
31: end for
```

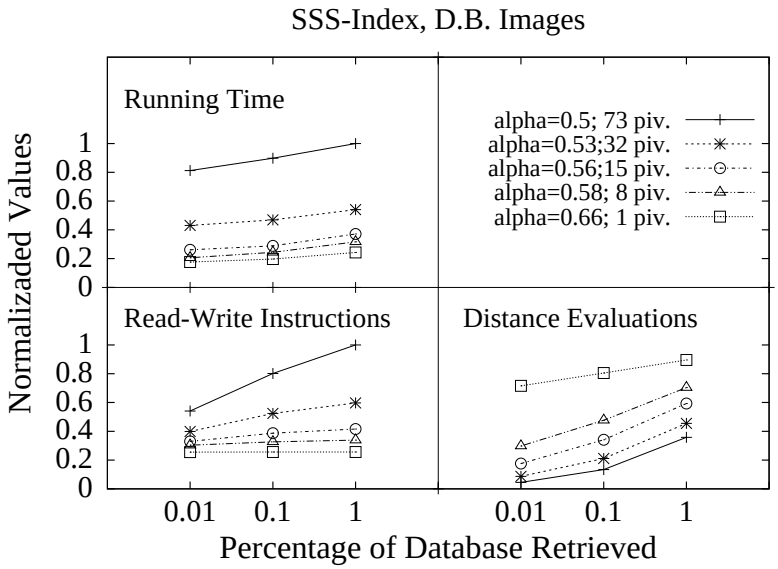


Figure 4.1: Range search algorithm using *SSS-Index* with different number of pivots.

In [8], authors have found empirically that $\alpha = 0.4$ yields the minimal number of distance evaluations. Our own experiments on GPU confirm this behavior: the more pivots are used (up to a certain threshold), the less distance evaluations are performed. However, as shown in Figure 4.1, the best performance in GPU is obtained with just one pivot. Indeed the more pivots used, the worst the execution time. *Irregularity* explains this apparent contradiction: when using more pivots, threads within a warp are more likely to diverge. Moreover, memory access pattern becomes more irregular and hardware cannot coalesce them. This leads to the observed increase in the number of read/write operations. In summary, less distance evaluations do not pay off due to the overheads caused by warp divergences and irregular access patterns. Overall, just one pivot provides the optimal performance for many of our reference databases.

We also implemented and compared an alternative strategy on the *SSS-Index*,

which distributes (circularly) the pivots among the threads, using the transposed matrix of distances.Thus, each thread try to discard (using triangle inequality) all the element of the database, using the distance from the query to its pivot. We write in shared memory which element can be discarded for any thread. But, because the size restrictions of that memory, this process is performed iteratively over a set of E elements each time. In each iteration, after testing E elements, a synchronization instruction is executed, and the non discarded elements are (circularly) distributed among the threads, and each thread performs the distance evaluation between the query and its assigned elements. The Table 4.1 shows the running time and quantity of reads/writes operations of this latter strategy (*Pivot-Distribution*) and the previous one, which distribute the elements among the threads (*Element-Distribution*), using the *Images* database.

The results show a low performance for the *Pivot-Distribution* strategy, reaching a running time up to 40 times of that reached by the *Element-Distribution* strategy when a radius that retrieves the 0.01% of the database is used. That is mainly because the high quantity of reads/writes operations performed. Note that, for the *Pivot-Distribution*, the running time decreases as the radius increases. This unexpected behavior may be explained again due to the irregular instruction flow. The larger the radius, the more regular the computation becomes.

4.1.4. Experimental Results to Process Range Queries on a single-GPU Environment

We have compared our GPU-based implementations against sequential and OpenMP based counterparts. As basis we have taken the proposals for implementation on

Table 4.1: Running Time (in seconds) and quantity of reads/writes to *device memory*.

Running Time (secs.)		
% Retrieved of the Database	***Pivot-Distribution***	***Element-Distribution***
0.01	88.8	2.2
0.1	82.8	2.4
1	76.4	3.0
Quantity of Reads/Writes (x10000)		
% Retrieved of the Database	***Pivot-Distribution***	***Element-Distribution***
0.01	455551	29699
0.1	461104	29740
1	468319	29806

multi-core systems introduced in Chapter 3. In that chapter we have shown that the best alternative for high query traffic is the so-called *Local* method, and that is the strategy used in the following experiments. The number of threads is always equal to the number of cores, and each thread is mapped onto a different core. The sequential implementation is based also on this code, but removing OpenMP primitives and pragmas (essentially the same implementation but with just one thread).

Our GPU experiments were carried out on a NVIDIA Tesla M2070 which is shipped with 14 multiprocessors, 32 functional units per multiprocessor, 48KB of shared memory and 5GB of device memory. The host CPU is a server, composed of 2 Intel's Nehalem processor Xeon E5645 with 24 GB of RAM, with a total of 12 processors, each of them with Intel Hyperthreading, resulting in 24 hardware threads. The sequential and multi-core algorithms were executed in this same machine. Table 4.2 details the hardware. Note that, even if the evaluated GPU architecture has a larger total number of functional units, their usage is restricted to SIMD-like computation: each function unit of a GPU multiprocessor executes the same instruction on different data every cycle. Thus, while the two evaluated archi-

Table 4.2: Main features of the computing platform for sequential and OpenMP implementations.

Processor	2xIntel Xeon E5645 (2.40 GHz)
L1 Cache	6x32KB + 6x32KB (inst. + data) 8-way associative, 64byte per line
L2 Cache	6x256KB 8-way associative, 64 byte per line
L3 Cache	12MB 16-way associative, 64 byte per line
Memory	24GBytes, (6x4GB) DIMM DRR3, 1333 MHz
Operating System	GNU Debian System Linux kernel 3.2.0-3-amd64 for 64 bits
Intel C/C++ Compiler v11.0 (icc)	-O3 -xW -ip -ipo Parallelization with OpenMP: -openmp

tectures are comparable from a technological point of view, the GPU is a throughput oriented architecture tailored to data level parallelism exploitation, while the Xeon server integrates larger caches and more sophisticated control that provides better performance for irregular computations.

We used the same databases of the previous Chapter 3 and we added one composed by high dimension elements, which are described below:

Words **:** A Spanish dictionary with 51,589 words. We used the *edit distance* [32] to measure similarity. We processed 40,000 queries selected from a sample of the Chilean Web which was taken from the todocl.cl search engine.

Images **:** A collection of images from a NASA database containing 40,700 images vectors, available in the Metric Space Library of sisap.org [25]. The queries were 23,831 elements. In this collection we used the *euclidean distance* to measure the similarity between two objects.

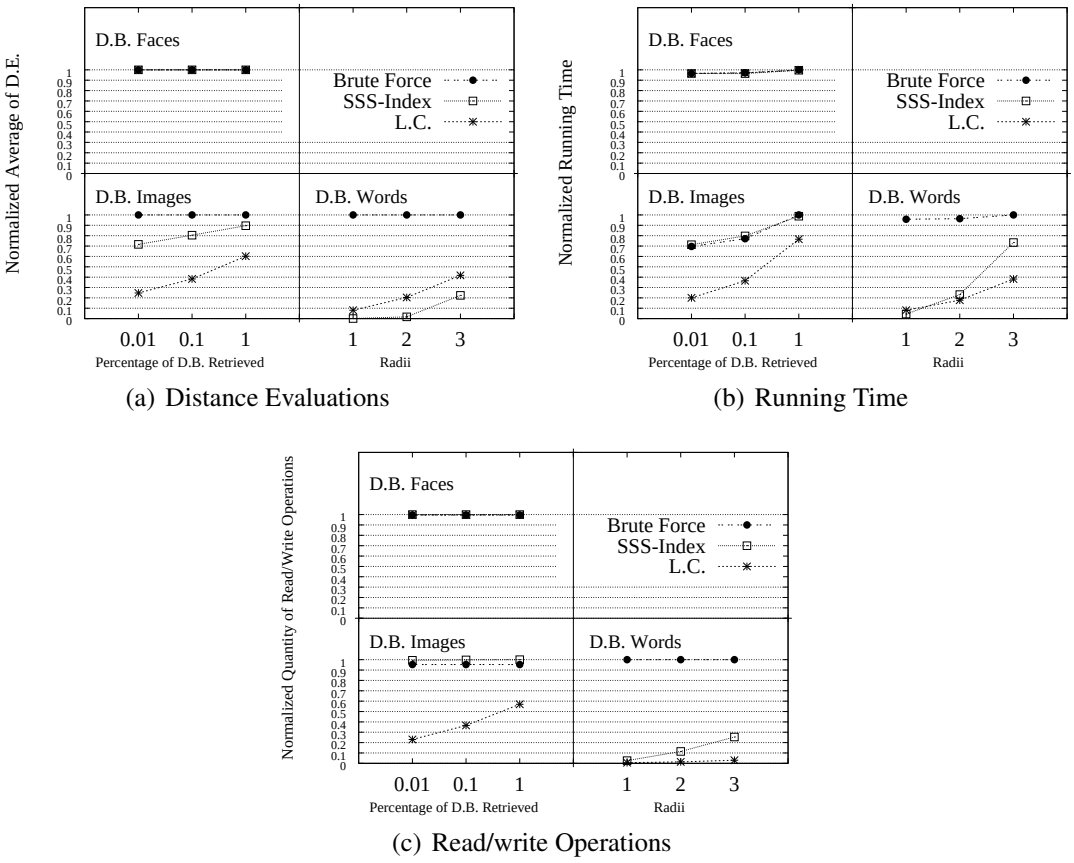
Faces **:** This database is a collection of 8,480 face images obtained from Face Recognition Grand Challenge ([50]). We apply the *Eigen Face Method* [54] to

obtain a projection matrix, that can be used to generate a feature vector from any face image. We used this collection as an empirical probability distribution, from which we generated a large collection of random face image objects, containing 95,325 objects of dimension 254. From the same dataset we took 23,831 elements as queries. We used the *euclidean distance* to measure the similarity between two objects.

In the vector databases (*Images* and *Faces*), the radii used were those that retrieve on average the 0.01%, 0.1% and 1% of elements from the database per query. In the *Words* database the radii were 1, 2 and 3. Similar values have been also used in previous papers [16, 41, 40]. In all the proposed methods, the set of queries are previously copied to device memory.

Regarding the GPU implementation, we performed a wide exploration to obtain the best parameters for each indexed structure. For *List of Cluster* (*LC*) we found that 64 elements per cluster is the best option for the vector databases (*Images* and *Faces*), while 32 elements per cluster performs the best in the *Words* database. We already discussed *SSS-Index* tuning in Section 4.1.3. The conclusions there drawn hold for the vector databases, so a single pivot is used. However, for the *Words* database it is better to enlarge the quantity of pivots, because the differences in read/write operations accessing lighter data of type `char`.

Figure 4.2 illustrates the performance characteristics of our GPU implementations. *Brute Force* stands for the exhaustive-search algorithm. *LC* and *SSS-Index* show the results for the two implemented indexing mechanisms with the parameters indicated above. All figures are normalized to the largest value of each experiment. This figure shows three graphs corresponding to the average of distance evaluations per query, running time and the number of read/write operations to device memory.



(a) Distance Evaluations

(b) Running Time

(c) Read/write Operations

Figure 4.2: Normalized values of the **a)** average of distance evaluations per query, **b)** running time, and **c)** read/write operations. Using a single GPU to process *range* queries.

We can observe that the performance behavior is not the same for all the databases. For the database *Faces*, the results are always the same, which is because all the methods, the brute force and indexing algorithms, access to the same elements of the database: all of them. The database *Faces* is composed by elements of high dimension, and as we explained in Section 2.3, the indexing methods are inefficient discarding elements on high dimension spaces. The *LC* and *SSS-Index* indexes are not able to discard elements, and they have to compare all the elements of the

database against the queries, which is what the brute force algorithm does.

Placing our attention on the results of Figure 4.2(a), as expected, indexing mechanisms do significantly decrease the number of distance evaluations in the *Words* and *Images* databases when compared to the brute force search method. In sequential computing the quantity of distance evaluations determines the behavior in the running time ([36, 24]) because its high associated cost, therefore one would expect that the running times sown by Figure 4.2(b) mimic the trend shown by the distance evaluations in Figure 4.2(a), but results in running time partially contradict this intuition, specially for the *Images* database. Figure 4.2(c) has the clue: memory access pattern, which heavily influences performance on current GPUs, behaves better for the *LC* index. As stated in Section 2.5, when a warp launches misaligned or non-consecutive memory accesses, the hardware is not able to coalesce it and a single reference may become several separate accesses. When activating the CUDA profiler, it can be seen that the number of read/write operations of the *SSS-Index* grows much higher than the *LC*, due to non-coalesced memory operations. This latter and the quite regular code of the *LC* explain its sustained superior performance over the other implementations.

Despite the read/write operations heavily influences on running time, there is an exception in the *Words* database with radius 1, where the *SSS-Index* obtains the lowest running time, but not the lowest quantity of read/write operations. The reason is because the number of read/write operations for that database and radius was small enough that other factors influenced, factors such as the amount of registers used by the threads in the *SSS-Index* was 12% lower, and because of this the number of active warps per multiprocessor is higher. Also the total number of instructions executed by the *SSS-Index* was 2% lower, and the number of warps that had to serialize its

access to shared memory (due to access conflicts to the same bank) was 34% lower in *SSS-Index*. But all these factors are irrelevant when the number of read/write operations increase, due to the high cost associated with these operations.

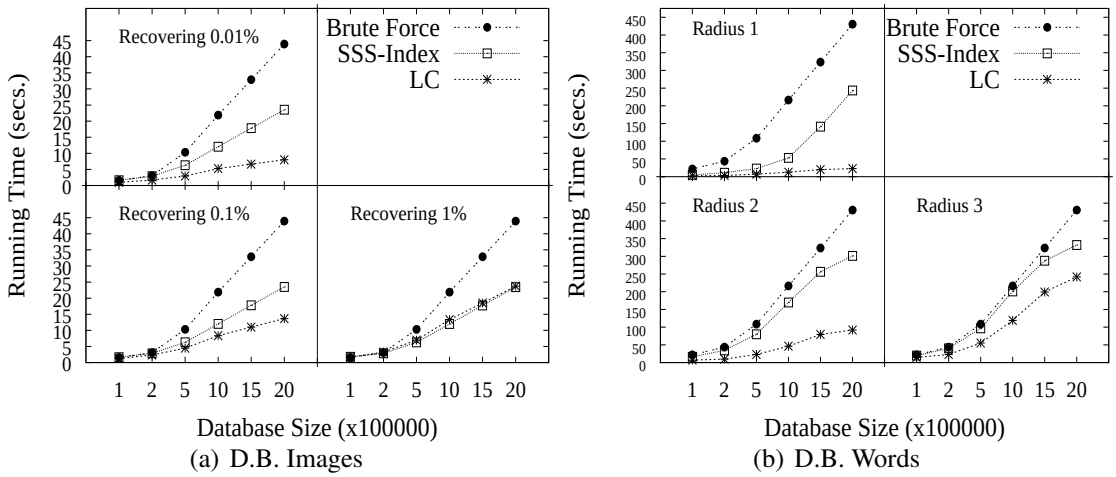
With regard to the results on the *Faces* database, we observe a different behavior. This is a high dimensional space, high enough for the indexes not be able to discard elements, being the *Brute Force* method the most efficient approach. The indexes try to discard elements but they cannot, therefore they compare all the elements of the database against the queries, which is exactly what the *Brute Force* method does, but the indexes execute more instructions applying the searching method of the index. This is the well-known *curse of dimensionality* ([5]): in high dimensional spaces the indexing methods in metric spaces lose efficiency.

We also changed the size of the databases to test the behavior of our algorithms when the number of elements of the database grows. The Figure 4.3 shows the GPU algorithms when the size of the database varies. In this experiment we used the following *extended datasets*:

***Images* :** We use the original database as a probability distribution to generate random vectors datasets of different sizes. We used the same euclidean distance as distance function and the same query log (23,831 queries) used with the original dataset.

***Words* :** Dictionary with words used in languages from UK. We used the same query log of 40,000 queries from todocl.cl used previously.

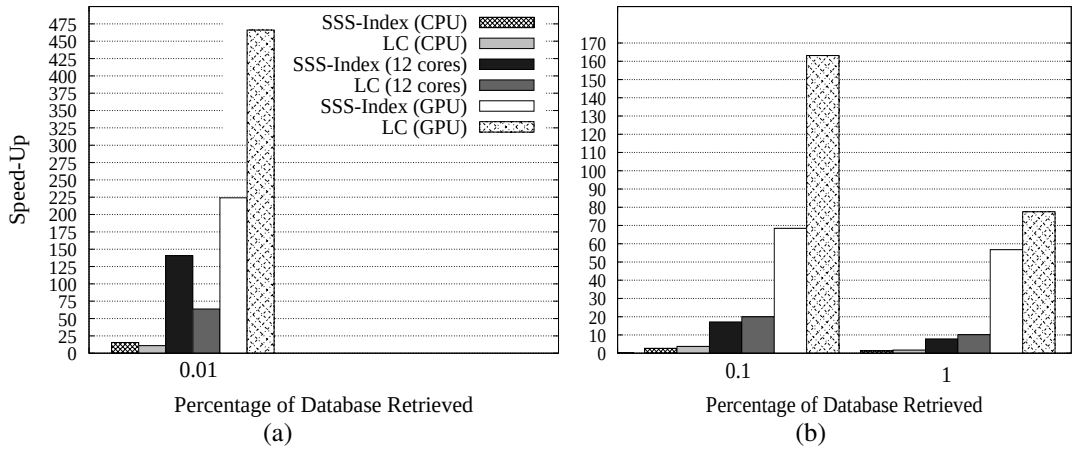
The least performance for both databases was shown by the *Brute Force* method, which calculates the distances between all the elements of the database and the queries. In the *Images* database (Figure 4.3(a)), the *SSS-Index* keeps its performance across the different radius, while the *LC* is affected when the percentage of



(a) D.B. Images (b) D.B. Words

Figure 4.3: Running time of single GPU methods varying the size of the database processing *range* queries.

recovering grows. But this latter shows the best performance recovering 0.01% and 0.1%, and is almost equal to *SSS-Index* recovering 1%. In the *Words* database the *LC* shows the best performance for all the radius.



(a) (b)

Figure 4.4: Speed-up of the *SSS-Index* and *LC* using different platforms (with *Words* database of 100,000 elements), over sequential brute force algorithm, processing *range* queries.

Focusing on overall performance, Figure 4.4 shows the speed-up of all our implementations taking as reference the performance of a sequential brute force version executed on a single CPU. In this experiment we used the *Words* database of 100,000 elements, but we got similar results with the *Images* database.

The first two columns of Figure 4.4 show the speed-up of the sequential implementations using the two indexes, *LC* and *SSS-Index*. The next two columns show the speed-up of the multi-core implementation on the Xeon server. Finally, the last two columns show the speed-up of our single GPU implementations described in Sections 4.1.2 and 4.1.3. Please note that we tune each implementation to attain the maximum performance, thus index parameters may vary across implementations. We run the search with three different ranges. Given the large speed-up variations, we separate the results in two figures.

Overall, the *LC* index on the GPU largely outperforms the rest, which was expected, given its good regularity and its access pattern to the device memory. One common and expected trend is that, the smaller the radius employed, the larger the performance benefits of the indexing (sequential or parallel) techniques. Remind that the reference case is the sequential exhaustive search (i.e. all the possible distance evaluations are performed). If we increase the range of the search, the number of distance evaluations gets increased for the indexed implementations, thus narrowing the gap with brute-force implementation. Despite the *SSS-Index* does not achieve the best performance, it is less affected than the *LC* when the radius is increased. That is mainly because the irregularity in the search algorithm of the *SSS-Index* (Algorithm 11) is not too much affected with an increased radius. Is enough that one thread of the same warp take a different path from the rest in the *if* sentence in line 26 of Algorithm 11 for adding irregularity.

For some readers, GPU speed-up factors may not look so impressive between our GPU with 14 multiprocessors on-board (and 448 cores), compared with the 12-cores Xeon based server used for OpenMP experiments. However, it is important to remind that each of this NVIDIA multiprocessor is extremely simpler than the Core/Nehalem microarchitecture based Intel CPUs; instruction level parallelism is almost not exploited while it represents the main source of performance for complex out-of-order processors.

We observed that OpenMP implementations scale worse than GPU versions when increasing the search radius. For the smallest radius (recovering 0.01%) the GPU implementation of *LC* gets a 466x of speed-up, against a 141x for the OpenMP based. If we increase the radius (recovering 0.1%), the GPU speed-up only reaches 163x but the benefits of the OpenMP implementation drops more deeply to 20x. For the biggest radio explored, the GPU still gets a 78x speed-up against a low 10x for the OpenMP counterpart. The common memory controller is a bottleneck for the multi-core sever, since accesses from the 24 hardware threads are issued concurrently. Conflicting accesses are then serialized, thus decreasing potential performance gains. Current graphic processor units overcome these limitations offering an enormous bandwidth between processing elements and the DDR memory. Access coalescing plays a crucial role in the right exploitation of this feature. Moreover, fine grained multithreading helps to partially hide the unavoidable and long memory latencies.

4.2. Processing k Nearest Neighbor (kNN) Queries on a single-GPU

Despite using the range query algorithms as basis to implement the algorithms to solve kNN queries, we found different difficulties to deal with the implementation and mapping of the indexes to solve kNN queries. It was mainly because the particular features of a GPU and its hierarchy memory.

In the following we describe our proposed methods of exhaustive and indexing search to solve kNN queries on a single GPU environment. As we did in the solution for range queries (Section 4.2), we also exploit here coarse and fine grained parallelism. The latter is seen when processing queries in parallel by different CUDA Blocks, and each query with a group of threads.

4.2.1. Exhaustive Search

In this section we propose two exhaustive search algorithms to solve kNN queries. The first is based on previous works ([10, 26, 30]) shown in Section 2.6.2: the distances of all the elements of the database against the query are calculated, and then these distances are sorted to select the first K elements as final result. In the second one, we propose that each query be processed with a different CUDA Block, and each CUDA Block uses a set of heaps [29] to find the K nearest elements, where the distances from all the elements of the database to the query are previously calculated.

Both previous algorithms receive as an input parameter the array of distances δ, where δ[i] is the distance between the i-th element and the query. To get the array

δ we must launch a kernel, where each thread calculates the distance between a different element of the database and the query. In the following subsections 4.2.1.1 and 4.2.1.2 we describe the sort-based and heap-based methods respectively.

4.2.1.1. Sort Based Processing

This method is based on sorting the elements of the array δ, which is received as an input parameter and contains the distances between all the elements of the database and the query. The final results are the K first elements of the sorted array. Effective implementations of this strategy on the GPU need an efficient parallel sort algorithm. In Section 2.6.2 we described the publications ([10, 26, 30]) that use this sort based method using the CUDA-based sorting methods *Radix-sort* [53] and *insertion sort* [26]. Cederman and Philips [11] proposed a *GPU-Quicksort* implementation that showed to be more efficient than previous GPU sorting methodologies. Because of the latter, we used the *GPU-Quicksort* to implement our sort based exhaustive search algorithm.

The *GPU-Quicksort* is divided in two phases: *partitioning* and *sorting*. The former consists of splitting the original vector into P partitions, that can then be sorted independently. This process is recursively repeated until there are enough partitions so that the GPU can be filled with a grid of CUDA Blocks, each CUDA Block being responsible of the sorting of one partition.

The partitioning process starts by selecting a global pivot; all the elements greater than this pivot are stored in one partition (high-partition) while the others are stored in the low-partition. Essentially, the original sequence is divided into M subsequences that will be processed by different CUDA Blocks. The implementation exploits the *atomicAdd* instruction (available in CUDA from computing capability

1.3), to make these CUDA Blocks cooperate in finding the correct positions, where they must store their respective elements, lower and greater than the global pivot found in the subsequence managed by the CUDA Block.

This *atomicAdd* instruction allows us to perform the partition of one sequence in a single kernel. For older devices, where no *atomicAdd* is available, this process requires three kernels, which introduces three synchronization barriers that limit the scalability of the implementation.

To obtain P partitions, at least $P-1$ pivots must be selected and the partition kernel (or kernels in case of no *atomicAdd*) must be executed $P-1$ times, which involves $P-1$ synchronization barriers.

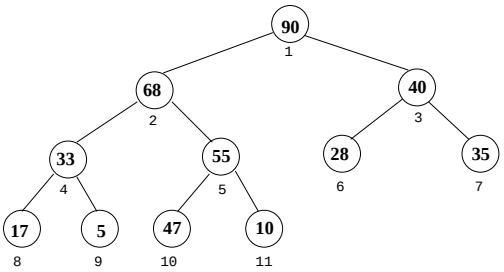
When the partitioning phase finishes, a grid of 1D blocks executes the sorting kernel. Each CUDA Block is in charge of sorting one partition. Sorting is performed in parallel by the threads in the CUDA Block using the same parallel-quicksort strategy, handling the recursion with an explicit stack implemented on shared memory. When the size of the subsequences generated by quicksort go below a given threshold, a *bitonic sort* [28] is used to complete the sorting.

Apart from the synchronizations needed by this algorithm, one main problem is the high probability to have unbalanced workloads for the different CUDA Blocks in the second phase. The size of the partitions depends on the pivots selected and cannot be controlled, unbalancing the work of the CUDA Blocks that are assigned partitions with different sizes.

4.2.1.2. Heaps Based Processing

In this section we propose a method where each CUDA Block processes a different query completely, using a set of *heaps* [29] in device memory to keep

the K nearest elements to the query across the search process. A heap is a binary tree, which allows store its elements in an array avoiding pointers. All the levels in a heap are always full, except maybe the bottom (see Figure 4.5). The elements in a heap are top-bottom sorted, thus the parent node has always higher (or less) key than its children, and the highest (or lowest) key is in the root node. To know what elements are the parents or children, is necessary follow the rules: (1) the children of the node $\{j\}$ are the nodes $\{2*j\}$ and $\{2*j+1\}$, (2) the parent of the node $\{k\}$ is the node $\{floor(k/2)\}$.



(a) Illustration of a heap in a tree.

90	68	40	33	55	28	35	17	5	47	10
1	2	3	4	5	6	7	8	9	10	11

(b) Representation of the heap in an array.

Figure 4.5: Example of a heap.

The Algorithm 12 shows the kernel used to process a query. Each thread of the i-th CUDA Block visits the elements of the array δ (according to a circular distribution) of the i-th query, keeping in its heap the K nearest elements to the query (line 2). For the latter is necessary to allocate in *device memory* a heap of size K per each thread of each CUDA Block. This set of heaps is stored in a matrix of $K \times T$, where T is the total number of threads (taking into account all the CUDA Blocks). Each column of this matrix represents a heap, in this way the i-th column stores the elements of the heap of the i-th thread.

After the CUDA Blocks has visited all the elements of the array δ, each thread has its K nearest elements to the query stored in its heap in device memory. Then a

Algorithm 12 Kernel of the reduction based on heaps.

{D is the dimension of the elements}
{tid is the ID of the thread inside the CUDA Block}
{δ is the array of distances between the elements of the database and the query}
{T_{Block} is the number of threads per CUDA Block}
{DH_i is the heap of the i-th thread stored in *device memory*}
{SH_i is the heap of the i-th thread stored in *shared memory*}
{$SIZE_{Warp}$ is the size of a *warp*}

Heap_Reduction(Elem δ, float *$Query$)

1: {Each i-th thread stores its K nearest elements to the query in its heap DH_i stored in *device memory*}
2: **for** ($i = tid$; $i < \delta$.size(); i += T_{Block}) **do**
3: $x.distance = \delta[i]$
4: $x.index = i$
5: insertion_heap(DH_{tid}, x)
6: **end for**
7: __syncthreads()
8: {A *warp* visits the elements of the heaps in DH and reduce them in $SIZE_{Warp}$ heaps stored in *shared memory*}
9: reduction_warp(DH, SH);
10: __syncthreads()
11: {A thread exhaustively visits SH and selects the first K elements as the final result}
12: **if** tid == 0 **then**
13: get_first_K(SH)
14: **end if**

reduction_warp(Heaps DH, Heaps SH)

if $tid < SIZE_{Warp}$ **then**
  for ($j = tid$; $j < T_{Block}$; j += $SIZE_{Warp}$) **do**
    for ($i = 0$; $i < K$; $i++$) **do**
      $x = DH_j$.pop()
      insertion_heap(SH_{tid}, x)
    end for
  end for
end if

insertion_heap(Heaps H, Elem x)

if H_{tid}.size()$< K$ **then**
  H_{tid}.push(x)
else
  if $x.distance < H_{tid}$.top() **then**
    H_{tid}.popush(x)
  end if
end if

reduction process is applied to find the final K results. This reduction is composed by two steps and it is illustrated in the Figure 4.6, where each triangle represents a heap. In the first step (function `reduction_warp()` in Algorithm 12), the elements of the heaps previously stored are distributed among the threads of the first *warp* (set of 32 threads) of each CUDA Block. But, each thread of the warp stores its K nearest elements in its heap stored in shared memory this time. In the second step, the first thread of the CUDA Block visits the elements of the heaps of the warp in the previous step (line 13), and finds the final K results storing them in a heap in shared memory.

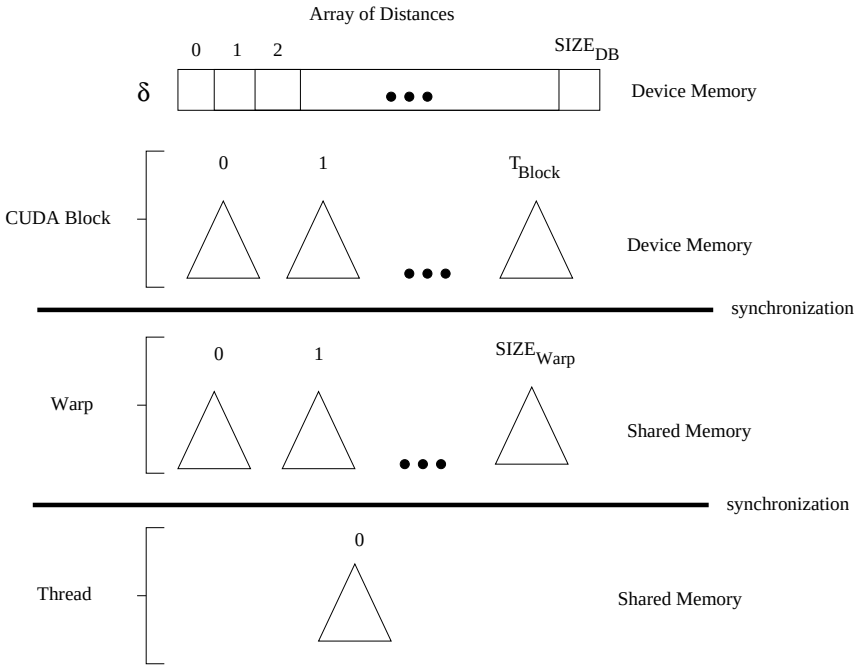


Figure 4.6: Illustration of the steps to reduce the array of distances δ to the final K results. Each triangle represents a heap.

The function `insertion_heap(H,x)` inserts a new element `x` in the heap `H`. This insertion is only made if the element is less than the root of the heap. Because

of always the root is extracted (`pop()`) and a new element inserted (`push(x)`), we defined the function `popush(x)`, which extracts the root and insert the new element in an optimized way.

It is noteworthy that the heaps of the same warp have their root elements in consecutive memory addresses. Thereby, we favor the coalescing of read and write operations.

4.2.2. List of Cluster (LC) on a single-GPU Processing *k*NN Queries

To process *k*NN queries with the *LC* we used the same structures used to process range queries (Section 4.1.2), i.e. three matrices named as *CENTERS*, *RC* and *CLUSTERS* in the Algorithm 13. *CENTERS* is a matrix of size $D \times SIZE_{cen}$ (D is the dimension of the elements and $SIZE_{cen}$ is the number of centers), where each column represents the center of a cluster. *RC* is an array that stores the covering radius of each cluster. *CLUSTERS* is a $D \times SIZE_{clu}$ matrix ($SIZE_{clu}$ is the number of elements in all the clusters) that holds the elements of each cluster. Index information is stored column-wise to favor coalesce memory accesses.

As explained in Section 2.1 there are two main methods to process *k*NN queries in sequential computing, based on range search. The first is the *decreasing range method*, which sets the range of the query in infinity, and after visit the first k elements, the range is continuously adjusted. That adjustment is made for each new visited element, thus the range search will be the distance to the k-th nearest element. The second one is the *increasing range method*, where the initial range of the query is set in a small value, and a range search is performed. If k results are not

found with the current range, it is increased, and a new range search is performed. The algorithm stops when k results are found.

The decreasing range method has shown better results in sequential computing than the increasing one ([52, 18]). But, we found that the increasing range method takes advantage when a GPU is used. The Figure 4.7 shows the running time, distance evaluations and quantity of read/write operations of the *LC* on GPU using the decreasing and increasing range methods over the *Images* database. The algorithm used in the experiments that implement the decreasing range method initialize the range in $r = \infty$, and this range decrease in two circumstances: (1) after the distances between the centers and the query are calculated, and (2) after each thread has performed a distance evaluation between an element and the query, which is done just if the distance is less than the current k-th nearest element. The adjustment is performed with the function `atomicMin(x,y)`, which blocks the access to `x`, then `x` is set in the minimum between `x` and `y`, and then unblock its access. On the other hand, the algorithm that implement the increasing range method sets an initial range (r_{ini}) with a small value and then a range search is performed. If k results are not found with the current range, it is increased, and a new range search is performed. This process is repeated until k results are found. In each new search the range is increased in Δ. We used the 1% of the database as a training set to define the values of the r_{ini} and Δ parameters.

The main reason of the low performance of the decreasing range method is because all the threads of a CUDA Block are involved in the solution of a query, and this intra-query parallelism does not allow to reduce fast enough the radius. In sequential computing, the decreasing range method adjusts the radius after visiting each element, but in the GPU version we have a set of hundreds of threads process-

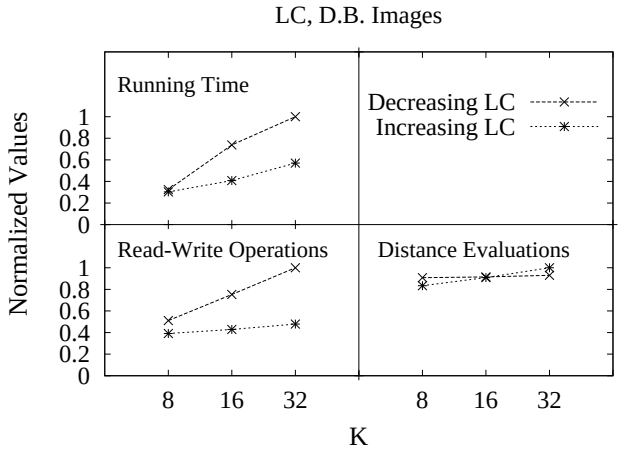


Figure 4.7: Normalized running time, distance evaluations and quantity of read/write operations of the decreasing and increasing range method to process *k*NN queries over the *Images* database with the *LC* index.

ing one query in parallel, therefore to keep a global radius implies a high number of synchronizations. On the other hand, the *increasing range method* adapts well to the features of the GPU for several reasons, such as: (1) the fact that the complete range searches are performed with the same range for all the threads, improves the regularity in the code, (2) the regularity in the code increase the coalescing of read/write operations, (3) this method implies less synchronization instructions than the decreasing method.

We also employ a set of heaps to implement the *LC* for keeping the *K* nearest elements to the query. If it is not possible to discard an element with the search method of the *LC*, that element is inserted in a heap (just if its distance to the query is less than that of the root of the heap). Thereby, after processing the query with the *LC*, we have one heap per each thread of the CUDA Block with their closest elements to the query. These elements are reduced to just one heap with the final results, using the same reduction steps used with the heap based method in the

previous Section 4.2.1.2.

The Algorithm 13 shows the code of the kernel that processes kNN queries using the *LC* index on a single GPU, using the increasing range method. The steps of the algorithm are delimited by the synchronization instruction `__syncthreads()`. In the first step the initial parameters are initialized and the query copied to shared memory (line 4). In the second step all the centers are distributed in a circular manner among the threads, and each thread calculates the distances between a subset of centers and the query (line 9), storing them in shared memory. In the third step, the elements of the clusters are assigned in a circular manner among the threads, and if the element belongs to a non-discarded cluster (line 16), the distance of the element against the query is calculated (line 17). If the element is inside the current range search, we try to insert it in the heap of the thread (line 19). The element is inserted in the heap just if the distance of the element to the query is lower than that of the root of the heap. After that, the centers are visited in a circular manner to check if some of them are inside the current range search (line 27). In the fourth and fifth steps, the first warp (line 35) and the first thread (line 39) of the CUDA Block reduce the heaps in just one, as it was shown in the previous Section 4.2.1.2.

4.2.3. SSS-Index on a single-GPU Processing kNN Queries

We represented this index with the same structures used to process range queries in Section 4.1.3, i.e. three matrices denoted as *PIVOTS*, *DISTANCES* and *DB* in the Algorithm 14. *PIVOTS* is a $D \times SIZE_{Piv}$ matrix (D is the dimension of the elements and $SIZE_{Piv}$ is the number of pivots) where each column represents a pivot.

Algorithm 13 Kernel of the *LC* to process *k*NN queries on a single GPU.

{*tid* is the ID of the thread inside the CUDA Block}
{*DH* and *SH* represent the set of heaps stored in *device* and *shared* memory respectively.}
{B_{Size} is number of elements of a cluster}
{$SIZE_{cen}$ is the number of centers of clusters}
{$SIZE_{clu}$ is the total number of elements in the clusters}

KNN_LC(float **CENTERS*, float **RC*, float ***CLUSTERS*, float **Query*)

1: __shared__ float *query*[*D*], *distC*[$SIZE_{cen}$], *range*=initial_range()
2: __shared__ int *minC*=$SIZE_{cen}$, *condition* = 1
3: **for** (*i* = *tid*; *i* <*D*; *i*+=T_{Block}) **do**
4: *query*[*i*] = *Query*[*i*]
5: **end for**
6: __syncthreads()
7: {The distances from all the centers to the query are calculated.}
8: **for** (*i* = *tid*; *i* < $SIZE_{cen}$; *i*+=T_{Block}) **do**
9: *distC*[*i*] = distance(*CENTERS*, *i*, *query*)
10: **end for**
11: __syncthreads()
12:
13: **while** *condition* **do**
14: **for** (*i* = *tid*; *i* < $SIZE_{clu}$ && (*i*/B_{Size})≤ *minC*; *i*+=T_{Block}) **do**
15: *indC* = *i*/B_{Size}
16: **if** *distC*[*indC*] ≤ *RC*[*indC*] + *range* **then**
17: **if** (*x.distance* = distance(*CLUSTERS*, *i*, *query*))< *range* **then**
18: *x.index* = *i*
19: insertion_heap(DH_{tid}, *x*)
20: **end if**
21: **end if**
22: **if** *distC*[*indC*]< *RC*[*indC*] - *range* **then**
23: atomicMin(*minC*, *indC*)
24: **end if**
25: **end for**
26: {If some center is inside the current range search}
27: **for** (*i* = *tid*; *i* < $SIZE_{cen}$; *i*+=T_{Block}) **do**
28: **if** (*x.distance*=*distC*[*i*])≤ *range* **then**
29: *x.index*=*i*
30: insertion_heap(DH_{tid}, *x*)
31: **end if**
32: **end for**
33: __syncthreads()
34: {A *warp* reduces the heaps in *DH* to $SIZE_{Warp}$ heaps stored in *shared memory*}
35: reduction_warp(*DH*, *SH*)
36: __syncthreads()
37: {A thread exhaustively visits *SH* and selects the first *K* elements as the final result}
38: **if** *tid* == 0 **then**
39: *quantity_results* = get_first_K(*SH*)
40: *condition* = evaluate_condition(*quantity_results*, *query*)
41: *range* += Δ
42: **end if**
43: __syncthreads()
44: **end while**

DISTANCES is a $SIZE_{Piv} \times SIZE_{DB}$ matrix ($SIZE_{DB}$ = number of elements of the database) where each element is the distance between a pivot and an element of the database. *DB* is a $D \times SIZE_{DB}$ matrix where each column represents an element of the database. The index information is stored column-wise to favor coalesce memory accesses. As in the *LC* (and for the same reasons), we used the increasing range method.

The Figure 4.8 shows the performance of the *SSS-Index* using the *Images* database varying the value of α. Like what happened with the processing of range queries in Section 4.1.3, we can observe that the more pivots used (up to a certain threshold), the less distance evaluations are performed. However, the best performance in running time is obtained with just one pivot. The reason is because when more pivots are used, the memory access pattern becomes more irregular and the GPU cannot coalesce read/write operations. Less distance evaluations do not pay off due to the overheads caused by warp divergences and irregular access patterns. Therefore in all the following experiments we use just one pivot for the vector databases. However, in the *Words* database, we empirically found 64 pivots is a suitable parameter. This latter difference in the database of words is mainly because: (1) the distance function (*edit distance* with words) implies much more irregularity in the code than the euclidean distance (used with vectors), worsening the coalescing of read/write operations, and (2) with the same transfer size (which can just be of 32, 64 or 128 bytes) in the words database we are able to transfer more elements because the size of a *char* compared with a *float*.

As in the previous sections, we also employ a set of heaps (one per thread of each CUDA Block) to implement the *SSS-Index* on GPU, for keeping the nearest elements to each query along the search. The Algorithm 14 shows the kernel used

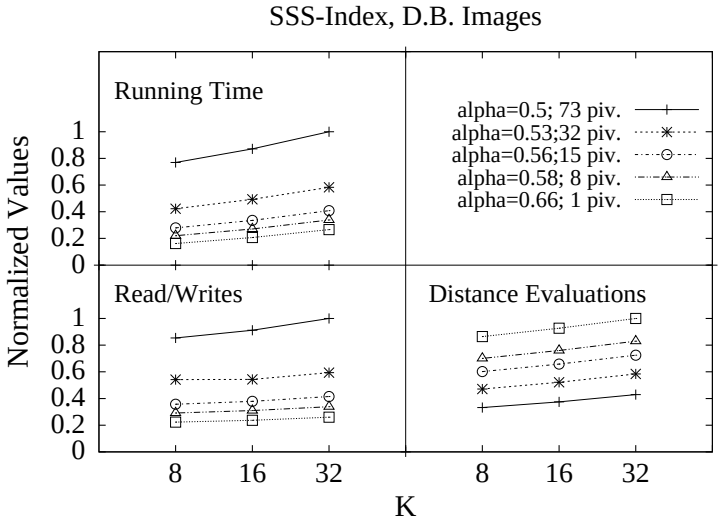


Figure 4.8: Normalized running time, quantity of read/write to *device memory*, and average of distance evaluations per query of the *SSS-Index* on a single GPU with the *Images* database.

by the *SSS-Index* to process *k*NN queries, which is divided in steps delimited by the synchronization function `__syncthreads()`. In the first step, the query is copied to shared memory (line 4). In the second step, the pivots are distributed (in a circular manner) among the threads and each thread calculates the distance between a subset of pivots and the query (line 9). In the case of using one pivot, just one thread perform this latter step. In the third step, the elements of the database are distributed in a circular way among the threads, and each thread try to discard its elements using triangle inequality (line 17). If the element cannot be discarded, then the distance between the element and the query is calculated by the same thread, and if it is the case, the element is inserted in the heap of the thread (line 25). An element is inserted in a heap just if its distance to the query is less than that of the root of the heap. In the fourth step, a reduction is applied (as in the *LC*) by the first warp (line 31). Finally, the first thread of the CUDA Block visits the elements of

the previous heaps and store the final K results in a heap stored in shared memory (line 35).

4.2.4. Experimental Results to Process kNN Queries on a single-GPU Environment

The experiments of this section were carried out on the same graphic card used for range query experiments (Section 4.1.4), i.e. a NVIDIA Tesla M2070 which is shipped with 14 multiprocessors, 32 functional units per multiprocessor, 48KB of shared memory and 5GB of device memory. The host CPU is a server, composed of 2 Intel's Nehalem processor Xeon E5645 with 24 GB of RAM, with a total of 12 processors, each of them with Intel Hyperthreading, resulting in 24 hardware threads. The sequential and multi-core algorithms were executed on the CPU host of the same machine (Table 4.2). Also, we used the same databases used in the range queries experiments: *Words*, *Images* and *Faces* which are described in Section 4.1.4.

4.2.4.1. Exhaustive Search Experiments

Figure 4.9 compares the different exhaustive search methods described in Section 4.2.1. *Sort-based Reduction* stands for the solution based on the state-of-the-art: all the distance evaluations are performed first. Next, the whole GPU is devoted to sort the obtained distances per query (i.e. queries are processed one at a time, and full resources are employed to sort a single vector of distances). A very efficient parallel version of the *quicksort* algorithm is employed at that step [11]. *Batch-Heap Reduction* corresponds with our proposal explained in Section 4.2.1.2: one CUDA Block solves a single query and multiple queries are solved in parallel. Finally, we

Algorithm 14 Kernel of the *SSS-Index* to process *k*NN queries on a single GPU.

{tid is the ID of the thread inside the CUDA Block}
{$SIZE_{DB}$ is the number of elements of the database}
{$SIZE_{Warp}$ is the number of threads of a *warp*}
{$SIZE_{piv}$ is the number of pivots}
{DH_i is the heap of the i-th thread stored in *device memory*}
{SH is the set of heaps of the first *warp* stored in *shared memory*}
{T_{Block} is the number of threads per CUDA Block}

KNN_SSS-Index(float **PIVOTS*, float **DISTANCES*, float **DB*, float **Query*, float *range*)

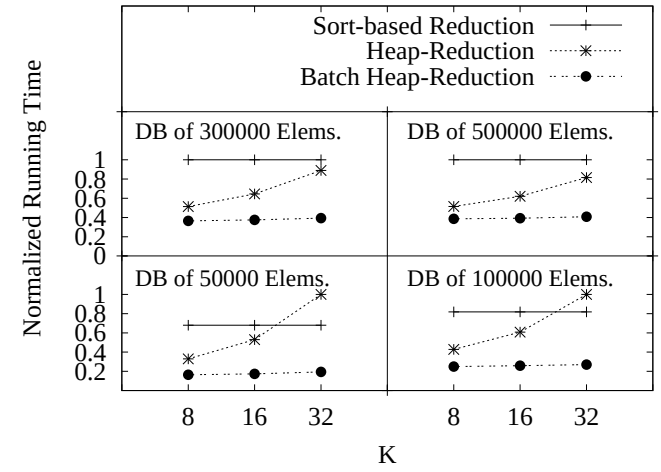
```
1: __shared__ float query[D], dist_piv[SIZE_piv]
2: __shared__ int condition = 1
3: for (i = tid; i < D; i+=T_Block) do
4:     query[i] = Query[i]
5: end for
6: __syncthreads()
7: {The distances between the pivots and the query are calculated}
8: for (i = tid; i < SIZE_piv; i+=T_Block) do
9:     dist_piv[i] = distance(PIVOTS, i, query)
10: end for
11: __syncthreads()
12:
13: while condition do
14:     for (j = tid; j < SIZE_BD; j+=T_Block) do
15:         discarded = 0
16:         for (i=0; i < SIZE_piv; i++) do
17:             if dist_piv[i] < DISTANCES[i][j] - range || dist_piv[i] > DISTANCES[i][j] + range
                then
18:                 discarded = 1
19:                 break
20:             end if
21:         end for
22:         if discarded == 0 then
23:             if (x.distance=distance(DB, j, query)) <= range then
24:                 x.index=i
25:                 insertion_heap(DH_tid, x)
26:             end if
27:         end if
28:     end for
29:     __syncthreads()
30:     {A warp reduces the heaps in DH to SIZE_Warp heaps stored in shared memory}
31:     reduction_warp(DH, SH)
32:     __syncthreads()
33:     {A thread exhaustively visits SH and selects the first K elements as the final result}
34:     if tid == 0 then
35:         quantity_results = get_first_K(SH)
36:         condition = evaluate_condition(quantity_results, query)
37:         range += Δ
38:     end if
39:     __syncthreads()
40: end while
```

include a third version labeled *Heap-Reduction*, which is similar to our previous method based on heaps, but it solves just one query at a time, using just one CUDA Block. This latter method was added to observe how our proposal scale from using just one CUDA Block to use all of them (in the *Batch-Heap Reduction* method). Figure 4.9(a) compares normalized running times for different reference database sizes and different values of K. The points are normalized to the largest value of the experiment. Figure 4.9(b) shows the cumulative running time of the same set of experiments. Both experiments were performed using the high dimension *Faces* database, because the exhaustive search algorithms have shown competitive results on high dimensional spaces ([17]).

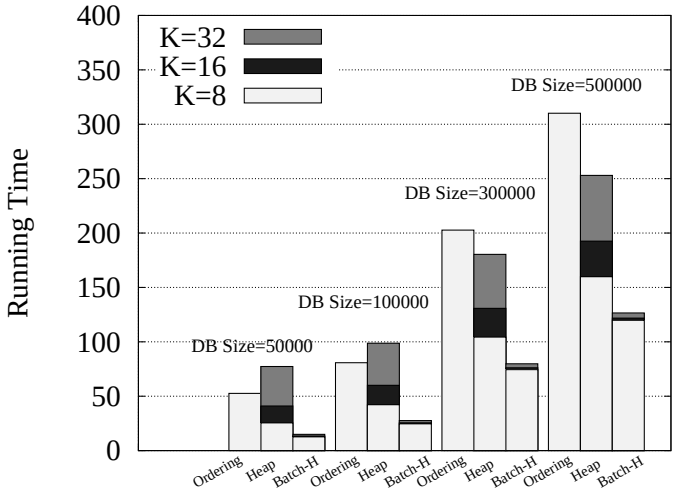
Our proposals are able to outperform the sort-based reduction counterpart in most experiments. Even with our *Heap-Reduction* method, which solve one query at a time using just one CUDA Block, we outperform the sort-based algorithm for large databases and small values of K due to a better memory management. When we exploit the full strength of GPU launching as many CUDA Blocks as queries the difference increases and, more relevant, our implementation becomes much less sensitive to K. Note the performance of any sort-based implementation is independent of K, since they sort the full distance vector.

4.2.4.2. Indexing Search Experiments

We now turn our attention to the proposed indexing algorithms to process *k*NN queries, *LC* (Section 4.2.2), and *SSS-Index* (Section 4.2.3). We empirically found for the *LC* that 64 elements per cluster in the vector databases and 32 in the *Words* database are suitable parameters. Similarly, we restrict ourselves to the increasing radius approach since it always perform better than the decreasing counterpart (as



(a)



(b)

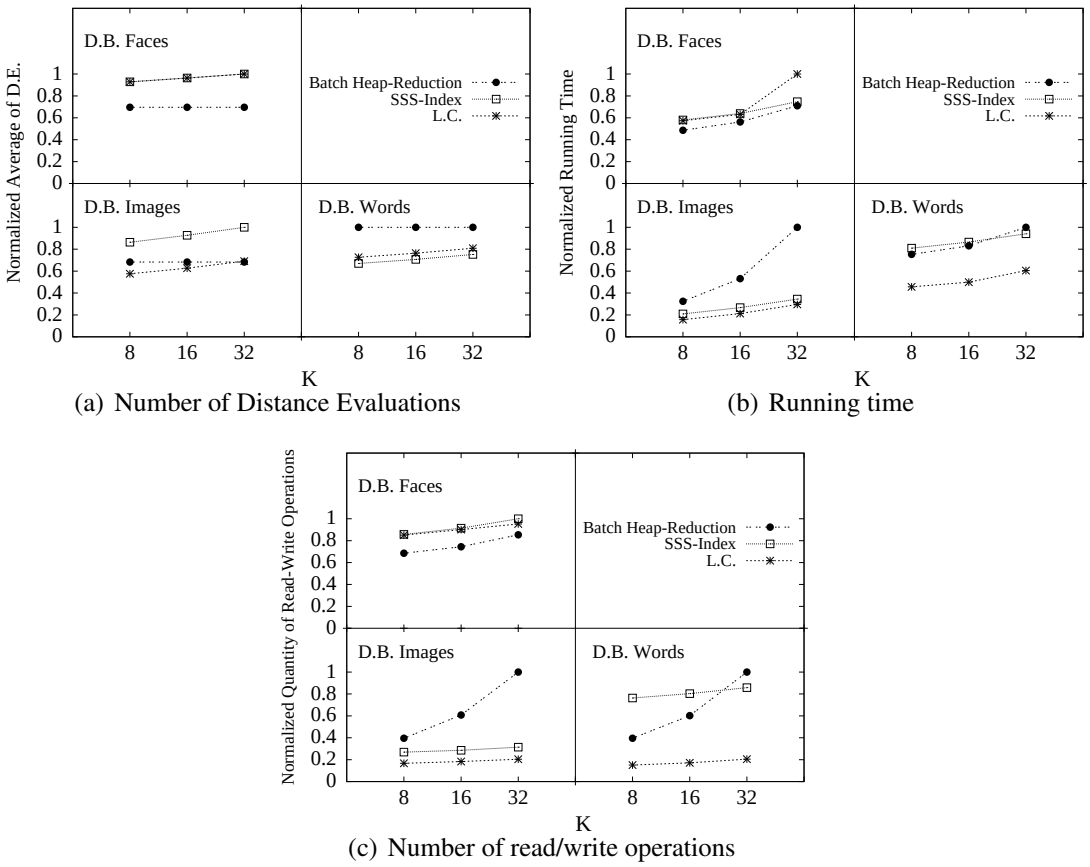
Figure 4.9: Normalized (a) and absolute (b) running times of the investigated exhaustive search algorithms for different K and number of elements using *Faces* database.

we mentioned in Section 4.2.2). Regarding *SSS-Index*, and following the conclusions drawn in Section 4.2.3, we use just a single pivot ($\alpha = 0.66$) for vectors databases and 64 pivots ($\alpha = 0.5$) for *Words* database.

Figure 4.10 shows different set of results to illustrate several important findings

of our three implementations. *Batch Heap-Reduction* stands for the exhaustive search algorithm. *LC* and *SSS-Index* show the results for the two implemented indexing mechanisms. All figures are normalized to the largest value of each version. We first place our attention on the total number of distance evaluations (Figure 4.10(a)). The *Words* database behaves as expected: indexing mechanisms do significantly decrease the number of distance evaluations when compared to the exhaustive search method. However, with the databases of vectors, that is no longer the case. Indeed the opposite behavior is observed: indexing mechanisms perform more distance evaluations than the exhaustive algorithm. This is specially true for *SSS-Index* with just one pivot. Obviously, this fact implies that some evaluations are performed more than once with the indexing mechanism, which is possible due to the increasing range approach. It must be noted that we intentionally decided not to reuse distance evaluations to avoid repeated computations since it introduces an enormous source of irregularity. On GPUs, decreasing the amount of work in this way, does not pay off.

We observed in Section 4.1.4 solving range queries that, the running times do not mimic the trend exhibited by the distance evaluations, and the same occurs in these experiments solving kNN queries. Memory access pattern and quantity of read/write operations (Figure 4.10(c)), which heavily influences performance on current GPUs, behaves better for the indexing mechanism, specially for *LC*. As stated in Section 2.5, when a warp launches misaligned or non-consecutive memory accesses, hardware is not able to coalesce it and a single reference may become up to 32 separate accesses. In all our implementations, heap insertions usually imply warp divergences and lack of locality, thus increasing the number of read/write operations. Indexed algorithms perform more distance evaluations but, since many



(a) Number of Distance Evaluations

(b) Running time

(c) Number of read/write operations

Figure 4.10: Normalized **a)** Distance evaluations per query (average) **b)** Running time and **c)** Read-write Operations (of 32, 64 o 128 bytes) to *device memory*.

distance evaluations are evaluated several times, the number of heap insertions is significantly reduced. However, as dimensionality increases the higher cost of these evaluations starts to trade-off the difference in heap insertions. There, indexed mechanisms perform poorly and our exhaustive-search implementation outperforms both of them. The *Faces* database, the one with largest dimensionality in our experiments, illustrates this situation (see Figure 4.10(b)).

Figure 4.11 shows the speed-up performance of: (1) our indexed implementations in GPU; (2) a multi-core version where each thread solve its queries with no

communication with the rest of threads (following the *Local* strategy describe in Chapter 3) and using a decreasing range method; (3) a sequential version of the indexes. All the speed-ups were obtained over a sequential brute force algorithm, which use a heap as auxiliar structure to keep the closest elements to the query. The results reinforce those shown by Figure 4.10, where the *LC* index achieves the best performance, achieving up to 42x, considerably outperforming the multi-core version for both indexes.

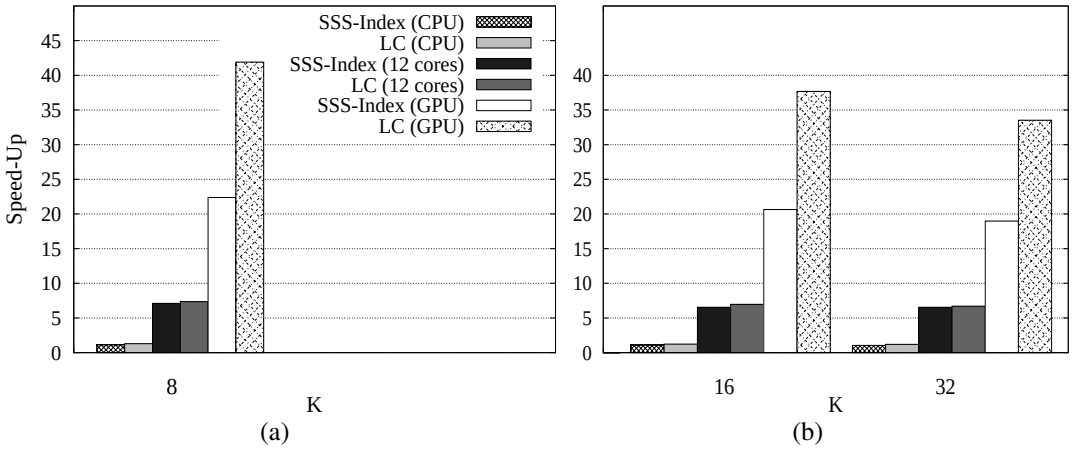
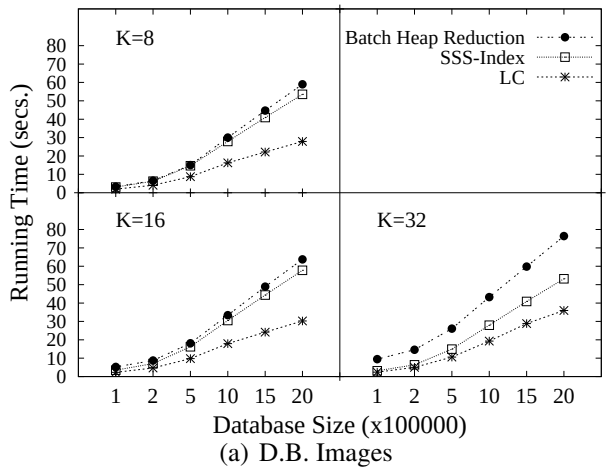


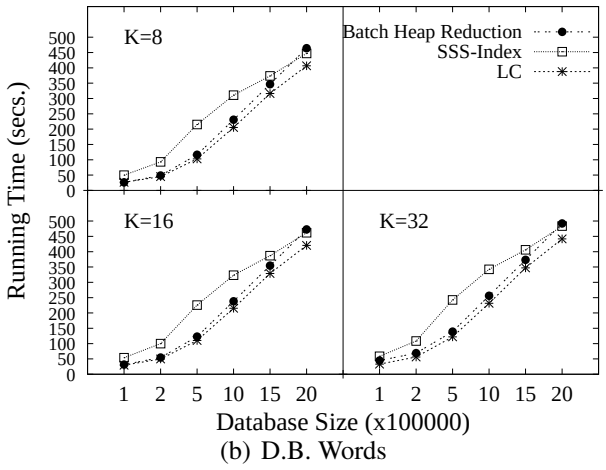
Figure 4.11: Speed-up of the *SSS-Index* and *LC* using different platforms (with *Words* database of 100,000 elements), over sequential brute force algorithm, processing *k*NN queries.

Figure 4.12 shows the running time for the indexing and exhaustive search methods varying the size of the databases. We used the same extended *Images* and *Words* databases used in the experiments for range queries (Section 4.1.4). In both databases, the *LC* achieves the best performance for all the different sizes. In the *Words* database, we can see that the exhaustive method outperforms the *SSS-Index* with small sizes. The reason for that is because the increasing method

implies to repeat the search each time that k results are not found, and each new search is performed with an increased range. But, due to the distance function (*edit distance*) is discrete, the quantity of recovered elements is very sensitive to the increasing of the range, adding irregularity in the code and worsening the coalescing of read/write operations. For the latter, the increasing range method is better suited with a continuous distance function, as we also could observe in the Figure 4.10.



(a) D.B. Images



(b) D.B. Words

Figure 4.12: Running time of single GPU methods varying the size of the database processing kNN queries.

4.3. Conclusions

In this chapter we proposed and compared CUDA based algorithms on a single GPU to process *range* and *k*NN queries in metric spaces. We used in this chapter the metric indexes *LC* and *SSS-Index* because: (1) their good results observed in the previous Chapter 3, and (2) work on dense matrices, and (3) exhibit certain regularity in the access pattern. All these features are very suitable to improve coalescing of memory access in the GPU. In our exploration we have found that some optimum parameters in GPU are very different of those used in sequential computing, in particular with the *SSS-Index* the optimum is reached with just one pivot for vector databases.

Due to the complexity and restrictions of the GPU, we found different problems for both kinds of queries, *range* and *k*NN queries, thus we applied different parallelization strategies for each type.

With regarding to our proposals to solve *k*NN queries, they used a set of heaps stored in device memory to keep the K nearest elements to the query across the search process. Afterwards, a *warp* and the first thread of the CUDA Block reduce the elements of the heaps in just one heap stored in shared memory with the final results of the query. We outperformed the previous methods of the related work based on sorting. Also, we showed that the *increasing range* method is more suitable on a GPU environment than the decreasing one, which is opposite to what occurs in sequential computing.

In both kind of queries, the *LC* index reached the best performance given its good regularity and its access pattern to the device memory. It achieves up to 466x of speed-up over the sequential brute force algorithm.

Chapter 5

Distribution and Search Strategies on a multi-GPU platform

Due to the good results and high speed-up achieved with the single-GPU strategies in the previous chapter, we extended them to a multi-GPU platform. But, in this chapter we use just the *LC* index because it achieved better results on a single-GPU platform.

We propose and compare different distribution strategies of the *LC* across the GPUs. In [27] and [37] are shown several strategies to distribute the *LC* on a clusters of processors connected by an Infiniband 1000 MB/s network. Due to the differences between the latter platform and the multi-GPU server, the same results cannot be applied in this case. But, the work shown on those publications was taken as basis for this chapter.

We divided this chapter in two sections. In Section 5.1 we assume that the database fits completely in device memory (GPU memory), therefore the multi-GPU strategies are focused on the offline distribution of the data and the processing

of the queries on those (fixed in memory) data. In Section 5.2 we assume that the database does not fit in device memory, therefore we focus on multi-GPU strategies for efficient replacement of the data to maintain a high throughput when processing queries.

5.1. Case 1: Database Fits in Memory

In this section, we propose and compare two multi-GPU strategies, assuming that all the database has been previously loaded in device memory. We are able to manage all the GPUs memories, and this allows to store a bigger database. To exploit this property, we just took into account strategies that create a global index with no replication, distributed among the device memories of the GPUs. Below we present and compare two different strategies, called *2-Stages* and *1-Stage* strategies. *2-Stages* strategy processes the queries in two different steps: in the first step we go over all clusters to filter those that can be completely discarded; in the second step the distances of the non-discarded clusters against the queries are calculated. On the other hand, *1-Stage* strategy just use one step, which gets the non-discarded clusters that must be compared against the queries, and performs the distance evaluations of those clusters and the query in the same step. *2-Stages* strategy has the advantage of making a better distribution of the non-discarded cluster among the GPUs, but *1-Stage* implies less synchronization functions.

5.1.1. 2-Stages Strategy

This strategy assumes that we process the queries in batches, and the main idea is to divide the search process of the query batches in two steps: (1) the first

step process in GPU which clusters cannot be discarded for each query, and the information result (which is transferred back to CPU memory), is used as an input parameter for the second step, (2) where the distances between the corresponding clusters and the queries are calculated in GPU.

In order that any GPU could decide which clusters must be compared with a particular query, all the GPUs have a copy of all the centers ($C_1, C_2, ..., C_N$ in Figure 5.1) and its covering radius. The elements of the clusters represented as $Cluster_1, ..., Cluster_N$ in Figure 5.1 are distributed in a circular manner. All the GPUs have a map that indicates the ID of the GPU where each cluster is stored. Because the latter, it is required to send a query to just one GPU.

In the first step (called *Setting Scheduling* in Figure 5.1), the queries are evenly distributed across GPUs. All the threads of a CUDA Block cooperate to solve a single query. In this step, the centers are distributed in a circular manner among the threads of the CUDA Block, and each thread determines which clusters can be discarded. This would correspond to the process made by lines 12-21 of the (1-GPU) Algorithm 10 to process range queries in Section 4.1.2. After that, each GPU outputs a matrix which indicates, for each query, which clusters must be further examined and thus which GPUs must be consulted to finish that query processing. That matrix must be stored in device memory because it is an input parameter for the next stage, executed in other kernel.

In the second step (called *Applying Scheduling* in Figure 5.1), the CPU processes all the matrices to set up separate *work queues* for each GPU. The *i*-th work queue of a GPU contains the list of every query that needs to be compared against some clusters held by the *i*-th GPU. An element of a work queue is a tuple $<$*query, list of candidate clusters*$>$, so it specifies, which clusters of that GPU have to be examined

to finalize that query. This would correspond to the lines 26-32 of the (1-GPU) Algorithm 10. Note that it is not required that every query resides in all the queues: in the best case it will just lay in one single queue if all the clusters relevant for the query are held by the corresponding GPU.

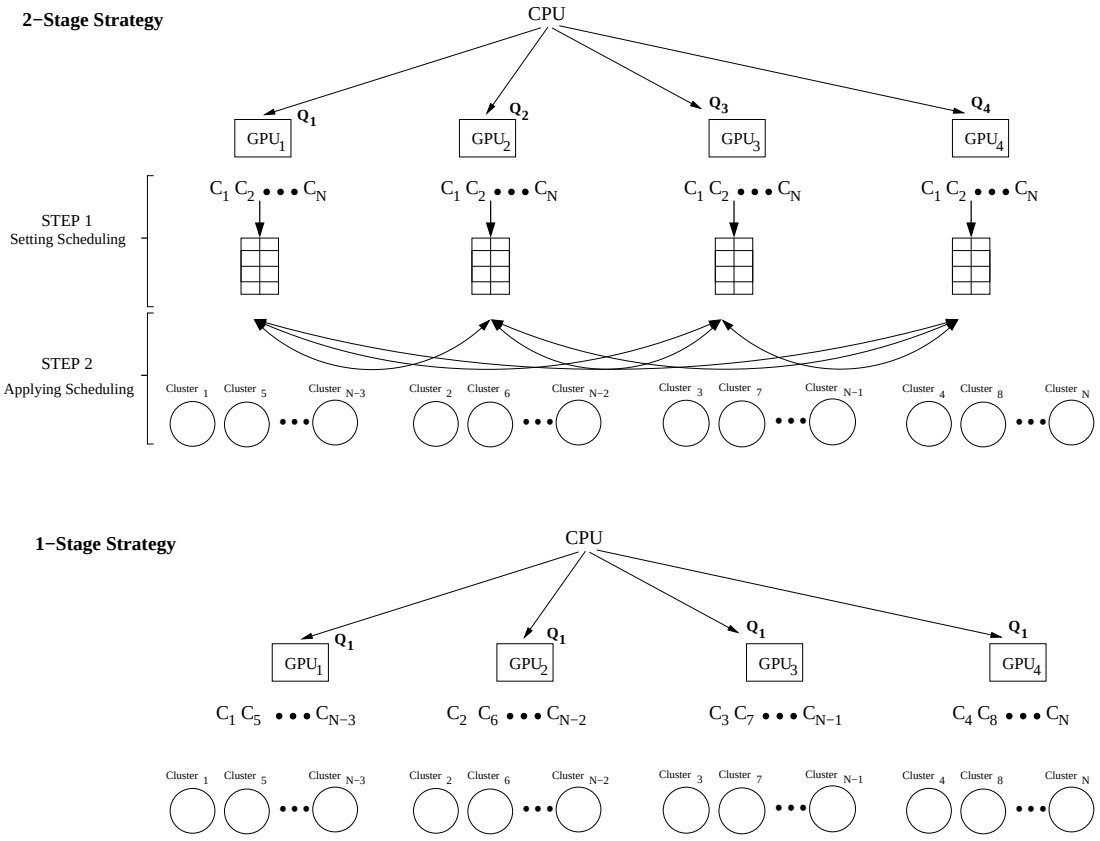


Figure 5.1: Illustration of the multi-GPU strategies *2-Stages* and *1-Stage*.

5.1.2. 1-Stage Strategy

The aim is to solve each query in just one step, and to avoid making a scheduling for each query. As we can see in Figure 5.1, the centers $C_1, C_2, ..., C_N$ and their respective element of clusters ($Cluster_1, ..., Cluster_N$) are distributed among the GPUs, i.e. every cluster is completely allocated in only one GPU, thus no index

information is replicated across GPUs. Because of the latter, each query must be sent to all the GPUs to be processed.

After a query is sent to all the GPUs, each GPU proceeds just as for the single-GPU *LC* implementation (Section 4.1.2). Each query is thus fully solved in a single kernel launch, and many queries may be launched in parallel.

One advantage of this strategy is that it reduces the number of kernel invocations and accesses to device memory, because the clusters that must be accessed are known in the same kernel launch. But, a disadvantage is that it is not as efficient as the *2-Stages* strategy to stop a searching when a query is contained into a cluster as allows the search algorithm of the *LC* index.

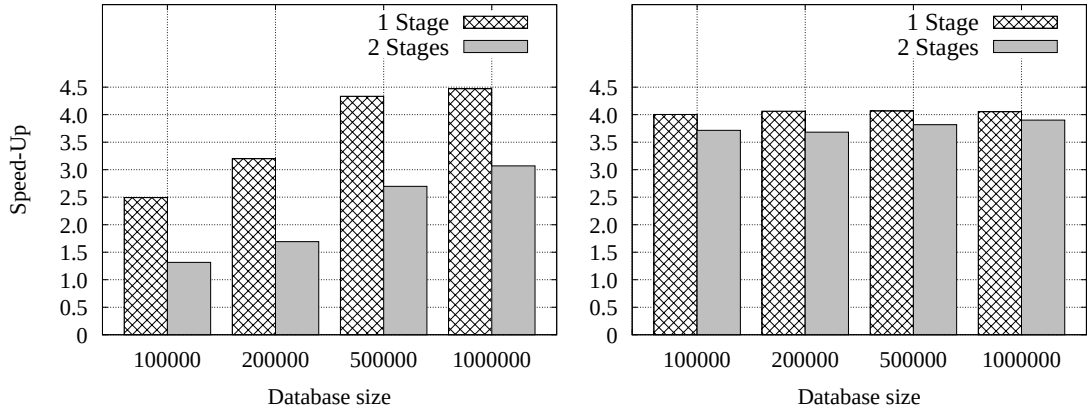
5.1.3. Experimental Results: Database Fits in Memory

In the experiments we use a multi-GPU server with 4 GPUs. Each GPU consists of a NVIDIA Tesla C1060 with 30 multiprocessors, 8 cores per multiprocessor, 16KB of shared memory and 4GB of device memory. We use the extended databases *Words* and *Images* described in Section 4.1.4.

2-Stage strategy is designed to better balance the work among the available nodes. Even if it succeeds, it introduces extra CPU-to-GPU communication that spoils the expected benefits. Moreover, due to the nature of the parallelization followed in the *1-Stage* strategy, there is no inter-GPU communication required and the only relevant communication penalty is the potential unneeded copies of queries to certain GPU nodes.

This explains the sustained better performance of *1-Stage* strategy over the *2-Stages* shown in Figure 5.2. This figure shows the speed-up of both multi-GPU

strategies over the single-GPU version of the *LC* index, where for fair comparison, the single-GPU version was executed on the same Tesla C1060 graphic card used for the multi-GPU versions.



(a) Images Database (b) Words Database

Figure 5.2: Speed-ups of the multi-GPU Strategies over the single-GPU version. All the versions including the baseline were executed under the same GPU card model (Tesla C1060).

The *1-Stage* strategy scales extremely well with database size and even super linear speedup (up to 4.5x speedup with just 4 GPUs) are observed for the *Images* database. This can be explained through the *occupancy* of each version. As stated in Section 4.1.2 (line 2 of Algorithm 10), the amount of shared memory required by our implementation is proportional to the quantity of centers stored in the node. For the single-GPU implementation, the whole index (i.e. the centers of all the clusters) is stored in the node, limiting the potential *occupancy* of the node up to 50%. Since the index itself is distributed across nodes, using 4 GPUs results in 100% *occupancy* and thus better exploits the available resources by launching more thread blocks concurrently, and allowing more active threads per multiprocessor.

We can also observe that, in the *Words* database (see Figure 5.2(b)), performance differences between the two strategies are significantly lower than in the *Images* database case. The higher cost of the distance evaluation function employed in the *Words* database minimizes the negative effects of extra synchronizations in the *2-Stage* strategy. Moreover, for the *Images* database, the first step of the *2-Stage* strategy is not able to balance the work among GPUs since most queries are finally dispatched to all GPUs in the last stage.

We also show the real execution times of our implementations on different platforms in Table 5.1 using the *Images* database.

Table 5.1: Real time in seconds for the *LC* on *Images* database (recovering 1% of the DB) using the different platforms. The query file was a log with 23831 queries.

LC	Multi-core (12 cores)	1 GPU (Tesla M2070)	4 GPUs (Tesla C1060) (1-Stage strategy)
100,000 elems.	4.8	1.6	0.64
200,000 elems.	11.2	3.24	1.01
500,000 elems.	32.2	9.16	2.11
1,000,000 elems.	68.6	17.74	3.97

5.1.3.1. Processing Queries in an on-line Environment

All results reported so far assume that we know all the queries to be solved in advanced. This means that we assumed that all the incoming queries are present in the system before we start solving them all in parallel. While this assumption could be admissible for certain use cases, it could be unaffordable for on-line *real-time* systems, like web searching for multimedia contents [38], where it is not possible to wait for thousands of queries before to start processing them.

A very important measure to take account, specially in web search engines, is the throughput, which is the number of queries solved per unit time. The Figure 5.3 shows the maximum throughput (number of queries solved per second) of the *1-Stage* strategy on the multi-GPU platform (the same multi-GPU platform described in Section 5.1.3), with all our databases and the three radii considered. For the *Images* database, a maximum of 80,794 queries per second is attained. It may still be low for high-traffic conditions but it is attained with just 4 GPUs and the proposed multi-GPU implementations scale pretty well when increasing both the number of GPUs and/or the size of the database.

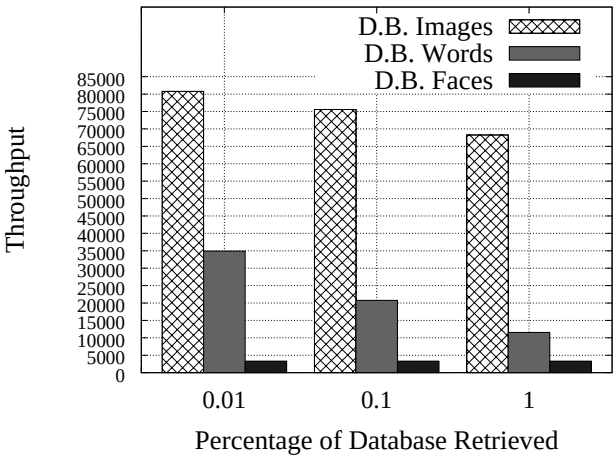


Figure 5.3: Throughput for *1-Stage* strategy (on the multi-GPU platform).

We also performed a second experiment measuring throughput, which is obtained in function of the number of queries issued in parallel. Figure 5.4 shows the results for the *LC* index using the multi-GPU platform of 4 GPUs, with the *1-Stage* strategy, retrieving 1% of the vectors databases, and using radius 3 for the databases of words. The x-axis indicates the quantity of queries that are processed in parallel (starting at five queries at a time), and the experiment finishes when all the elements

of the corresponding query log are processed. The y-axis shows the throughput of the system, expressed as the number of queries solved per second. The throughput rapidly increases up to the point where we launch 30 queries in parallel; this is due to the GPU used in the experiments, which includes 30 multiprocessors. Below that point the GPU is underused and the constant penalty of launching a kernel weights too much. There is a knee at 30 but throughput still increases very slowly due to the GPU multithreading capabilities which allows to hide long memory latencies effectively.

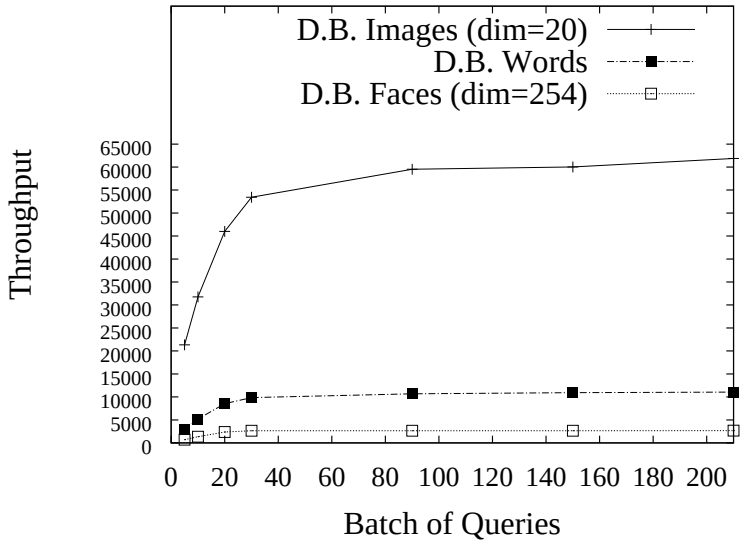


Figure 5.4: Throughput (queries solved per second) solving queries in batches, with *1-Stage* strategy over the multi-GPU platform (of 4 GPUs NVIDIA Tesla C1060), recovering the 1% of the data per query.

The *Images* database showed the best throughput, mainly because the low cost of a distance evaluation with elements of dimension 20, compared against elements of dimension 254 (*Faces* database). About the *Words* database, it shows a worse throughput than *Images*; again, this is due to the high cost of the distance evaluation

function, *edit distance*, which exhibits low regularity and variable memory access alignment.

Figure 5.5 reviews the speed-up for the *LC* implementation, using 4 GPUs with *1-Stage* strategy, with the *Images* database of 1 million of elements. Similar results were obtained with the database of words. This speed-up was calculated over the multi-core version of the *LC* (Chapter 3) with 12 cores, that knows all the queries in advance. The bar labeled *Maximized Batch* corresponds to the speed-up of our multi-GPU version of the *LC* when all the queries are known in advance. It is the upper bound for GPU performance, since all queries are solved with a single *kernel* invocation. The bar labeled *Batch=30* corresponds to our multi-GPU version of the *LC* with a scenario where, as soon as we have 30 queries in the system, we launch a kernel to solve them. In this experiment (23831 queries), this strategy implies 794 *kernel* invocations. Even solving queries in batches equal to the quantity of multiprocessors (30), our multi-GPU version achieves a competitive speed-up.

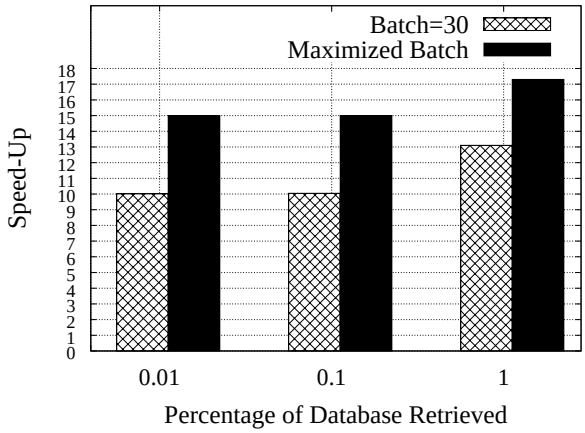


Figure 5.5: Speed-ups of the *1-Stage* strategy (using the multi-GPU platform of 4 NVIDIA Tesla C1060) over the multi-core version of the *LC* (with 12 cores), solving the queries in batches of 30 and with the maximum possible. The *Images* database of 1,000,000 elements was used.

We can see that the proposed multi-GPUs strategies can be used for *on-line* query processing in metric spaces for almost any traffic condition, representing a low-cost high performance alternative to traditional multi CPU implementations.

5.2. Case 2: Database does not Fit in Memory

This section proposes and compares different algorithms and strategies to solve similarity queries using databases large enough not to fit in device memory (GPU memory). The most real databases in production would fit on this case, where device memory is not enough to store all the data.

A critical factor in GPU are the CPU-GPU transfers, which heavily influence in the overall performance, therefore there is a need to efficiently schedule those transfers. In the following subections we propose a set of multi-GPU techniques to deal with the transfers of data between CPU memory and device memory (GPU main memory). Finally, we show that the combination of those techniques achieves the highest performance.

Specifically, in Subsection 5.2.1 we propose a hierarchical multi-level index with suitable properties for memory transfers in GPU. In Subsection 5.2.2 we propose a pipeline which makes use of CPU-cores and GPUs. In Subsection 5.2.3 we propose a second pipeline between memory transfers and kernel executions in the GPU. After that, in Subsection 5.2.4 we describe a method that combine all the previous techniques in just one, and finally in Subsection 5.2.5 we show the experimental results.

5.2.1. List of Superclusters (*LSC*) on GPU

We propose a hierarchical multi-level *LC*, named *List of Superclusters* (*LSC*) that takes into account the organization of the GPU memory.

The construction of the *LSC* has two steps. First, based on the construction procedure of the *LC* (Section 2.2.5) with fixed size of K elements, we get S clusters of size K. Each i-th cluster is composed by its center C_i, covering radius r_i and the K nearest elements to C_i ($k\text{NN}_{\mathbb{U}}(C_i, K)$). These S clusters are named *superclusters* and integrate the first level of the hierarchy. In the second step, we create a *LC* index into each supercluster with their own elements following the construction procedure of the *LC* (Section 2.2.5). The Figure 5.6 shows an example of a *LSC* index with two superclusters.

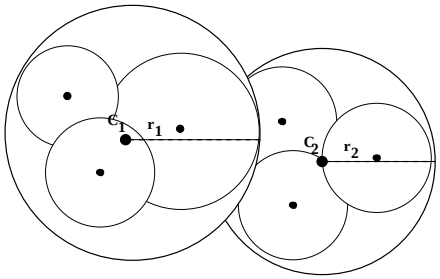


Figure 5.6: Illustration of the *LSC* index with two superclusters.

We set the parameters of the *LSC* to create versions with N and $N/2$ clusters per supercluster, where N is the maximum number of clusters allowed in device memory. Note that N is much less than the total number of clusters of the index, so multiple CPU to GPU transfers will be required to solve each batch of queries. It is noteworthy that we use a supercluster as the minimum unit transfer (between CPU and GPU memories) in all the following experiments that use the *LSC* index.

For searching in the *LSC* index, we first try to discard each supercluster, and for each non-discarded supercluster, we apply the searching procedure of the *LC* on the index inside the supercluster. Thus, an element will be compared against the query just if it belongs to a non-discarded supercluster and to a non-discarded cluster.

In GPU, after loading a complete supercluster in device memory, we launch a kernel with Q CUDA Blocks (Q is the quantity of queries of the current query batch) to search into it.We implemented the kernel that processes a range query (q,r) following the next three steps: (1) the first S threads cooperate to get the distance between the center of the supercluster and the query, where S is the quantity of superclusters; (2) we use those previous distance and the triangle inequality property to try to discard each supercluster C_i with covering radius r_i, i.e. if $d(C_i,q) \leq r_i + r$; (3) if the supercluster is not discarded, then we search in the *LC* index inside the supercluster, with the method used by 1-Stage strategy (Section 5.1.2).

5.2.2. Building a CPU-GPU Pipeline

To minimize the number of transfers to GPU and in order to increase the degree of parallelism, we developed a hybrid pipeline between CPU and GPU, where the CPU helps to discard some elements to avoid them to be transferred to the GPU. We use P CPU-threads, where P is the quantity of CPU-cores of the machine, and from those P the first G threads ($G < P$) manage a different GPU.

We implemented this pipeline for both *LC* and *LSC* indexes. Considering that N is the quantity of clusters that fit in device memory, and Q is the size of the current query batch, the steps of the pipeline (shown by the Figure 5.7), are as follows: (1) we try to discard N clusters of the *LC* (or superclusters in the *LSC*) with threads

in CPU, just using the center and covering radius of the clusters. For the latter we distribute (circularly) the centers among the threads, and each thread discards its cluster if its covering radius does not intersect with any of the Q queries. (2) We copy the ID of the non-discarded clusters according to the previous step. (3) We copy the non-discarded clusters to GPU memory and launch a kernel to process them in the GPUs with the corresponding Q queries. Taking account that we just need one CPU-thread to manage one GPU (step 3 of Figure 5.7), while the third step (with the first G threads) is in execution, the first step (with the remaining threads) is in execution too, but attempting to discard the next N clusters. As result, with this pipeline we load less quantity of clusters (or superclusters) in GPU, and minimize partially the penalty due to the limited GPU memory size.

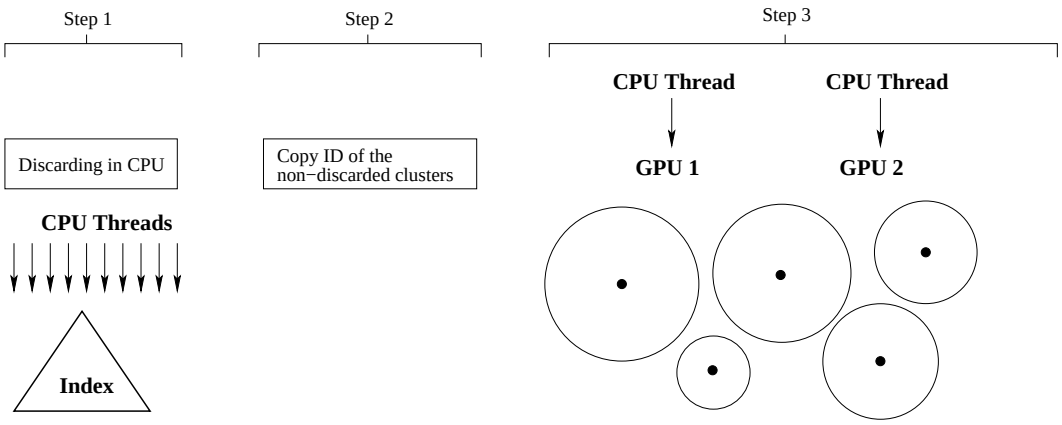


Figure 5.7: Scheme of the CPU-GPU pipeline.

5.2.3. Exploiting CUDA Asynchronous Copies

The function `cudaMemcpyAsync` allows to perform transfers to (and from) device memory while a kernel is in execution. This is possible by using *CUDA*

streams, where each CUDA stream can contain a different sequence of instructions. CPU to GPU transfers and kernels from different streams can be executed at the same time.

Modern NVIDIA's GPU (including the GPUs used in our experiments) have three engines in their hardware. One engine manages copies from CPU to GPU, the second manages the execution of kernels, and the third manages copies from GPU to CPU. The Figure 5.9 shows three cases of management of the different engines, where the first is the sequential case, i.e. just one CUDA stream is used. Depending on the model of the GPU, the codes of the Figure 5.8 can have a different effects illustrated by the Figure 5.9. This occurs in graphic cards that are able to concurrently run multiple kernels. When multiple kernels are issued back-to-back in different CUDA streams, the scheduler tries to enable concurrent execution of these kernels and as a result delays a signal that normally occurs after each kernel completion (which is responsible for kicking off the device-to-host transfer) until all kernels complete. So, while there is overlap between host-to-device transfers and kernel execution in the second version of our asynchronous code, there is no overlap between kernel execution and device-to-host transfers. The GPUs used in our experiments support running of multiple kernels, therefore we use a code like the *Asynchronous Version 1* of Figure 5.8.

Starting from the basic non-pipelined implementation, we exploit the asynchronous copies for both *LC* and *LSC* indexes, as the Figure A.22 shows. If N is the quantity of clusters that fit in device memory, then we create two CUDA streams, and each stream is composed by the following steps: (1) copy $N/2$ clusters to device memory, and in the case of the *LSC* copy one supercluster (which contains $N/2$ clusters), (2) launch a kernel to process the queries with the loaded clusters (or

Asynchronous Version 1:

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
  kernel<<x, y, z, i>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}
```

Asynchronous Verion 2:

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset],
                  streamBytes, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<x, y, z, i>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

Figure 5.8: Code of different approaches of pipeline between asynchronous copies and kernels.

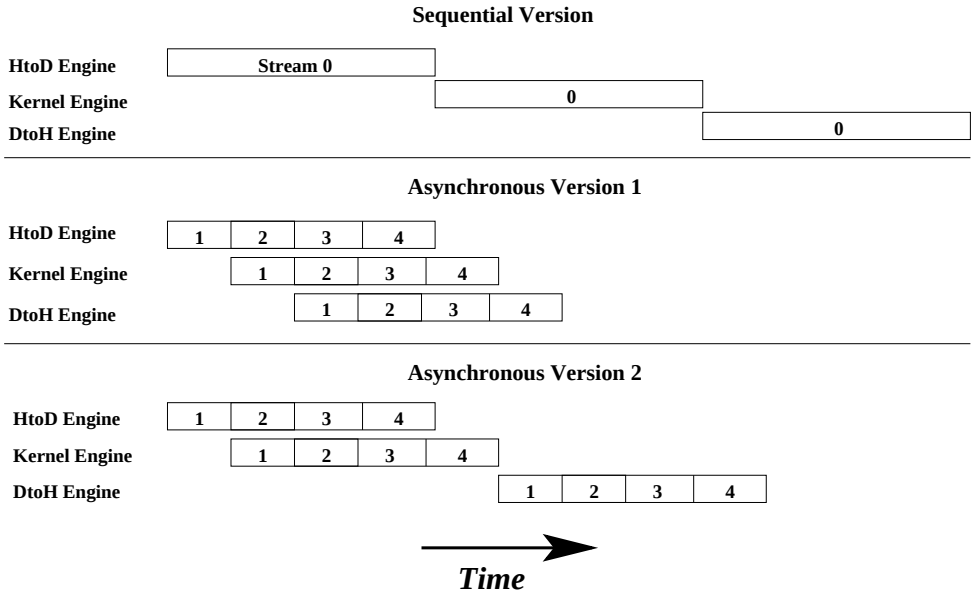


Figure 5.9: Illustrations of the codes shown by Figure 5.8, using four different CUDA streams.

supercluster). Then, CPU proceeds again with step 1 with the next set of clusters (or supeclusters) while GPU is with the step 2 processing the previous data set. We create just two CUDA streams and not more, because this quantity makes a good balance in running time between copies and kernels, which effectively builds a two stage transfer-kernel pipeline. In order to implement this pipeline, it is required to use page-locked (pinned) memory to transfer data.

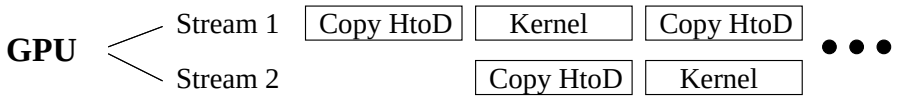


Figure 5.10: Scheme of the pipeline of asynchronous copies and kernels.

because their elements are contiguous in the database;

The CPU-GPU transfers are key to efficiently exploit the huge bandwidth between CPU and GPU. We always copy a cluster when using *LC* or a supercluster when using *LSC* with the minimum set of calls to the copy function `cudaMemcpyAsync`. The Figure 5.11 shows the matrix that stores the elements of clusters, which is column-wise of size $D \times E$ (D=dimension; E=number of elements), where the elements of the same cluster are contiguous. Because the latter, to transfer one cluster it is required to call D times to `cudaMemcpyAsync` (one for each row). We also have to call D times to the copy function to transfer one supercluster in the case of the *LSC*. Because the *LSC* covers a set of clusters, the *LC* has to call more times to the copy function, which is a disadvantage since short transfers could not hide the initial latency.

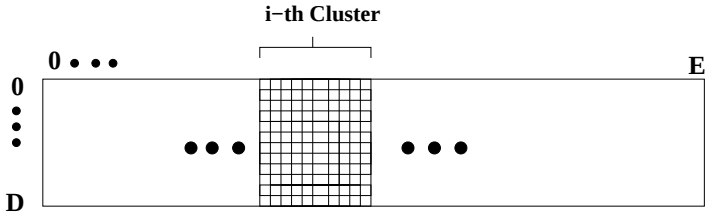


Figure 5.11: Scheme of the elements of a cluster stored in the matrix of elements of the *LC* index.

5.2.4. Multi-pipeline Strategy

Our final proposal combines the three previous strategies in one *multi-pipeline strategy*. We use the *LSC* index (Section 5.2.1), and we create P CPU threads, one per CPU-core, leaving G threads in charge of G GPUs ($G < P$). While the CPU-cores try to discard superclusters with the CPU-GPU pipeline described in Section 5.2.2, each GPU process non-discarded superclusters using the copy-kernel pipeline described in Section 5.2.3.

The Figure 5.12 shows a scheme of this strategy, which is composed by three steps separated by barriers, the steps are the following: (1) discard of superclusters with threads running on CPU-cores; (2) store the ID of the non-discarded superclusters in CPU memory; (3) the pipeline between asynchronous copies and kernel executions is applied, where a supercluster is transferred to device memory while a kernel is processing a previous supercluster. The steps 1 and 3 are executed at the same time.

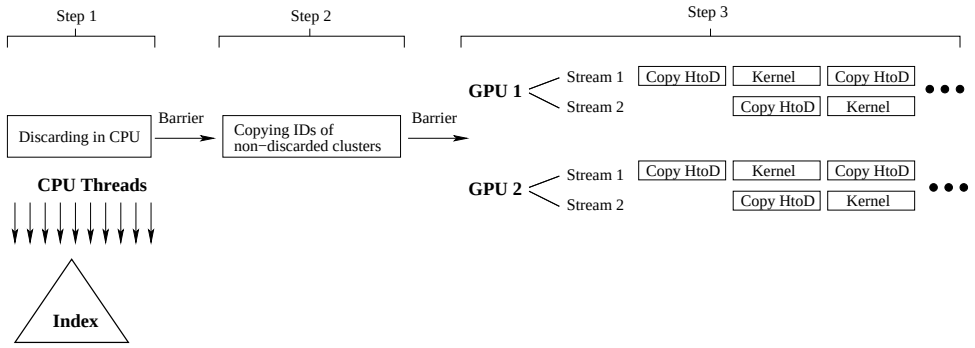


Figure 5.12: Scheme of the multi-pipeline strategy.

5.2.5. Experimental Results: Database does not Fit in Memory

All our experiments in this section are carried out on a multi-GPU platform composed by two NVIDIA Tesla M2070, and each one is shipped with 14 multiprocessors, 32 cores per multiprocessor, 48KB of shared memory and 5GB of device memory. The host CPU is a 2xIntel Xeon E5645 processor of 2.4GHz with 24 GB of RAM (detailed in Table 4.2).

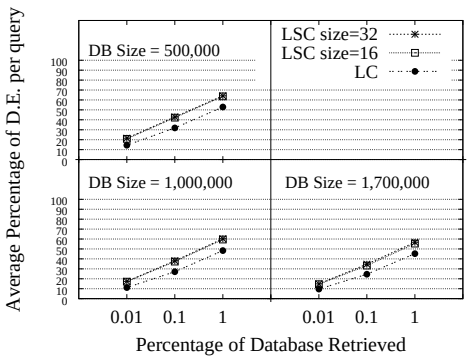
Due to the algorithms of this section are designed to deal with large databases similar to those used in production, we use as reference database the *CoPhIR* (Content-based Photo Image Retrieval) dataset [6]. This consists of metadata extracted from the Flickr photo sharing system. It is a collection of 106 million images containing, for each image, five MPEG-7 visual descriptors, specifically Scalable Color, Color Structure, Color Layout, Edge Histogram, and Homogeneous Texture. For the purposes of this thesis, we just used the *Color Structure* MPEG-7 image feature, which represents a 64 dimensional vector for each image. We use the *Euclidean distance* as a distance measure. The radii used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query.

To our best knowledge, there is not a public and real query log for similarity search in images. But recently, a public website was presented in [46]. It applies the MUFIN [59] search engine for images of CoPhIR dataset and it is used by many users all around the world. From this website we got our query log, which represents the queries processed along several days. We use 30,000 queries that are represented by its *Color Structure* MPEG-7 image feature of dimension 64. We have made this query log public in [1].

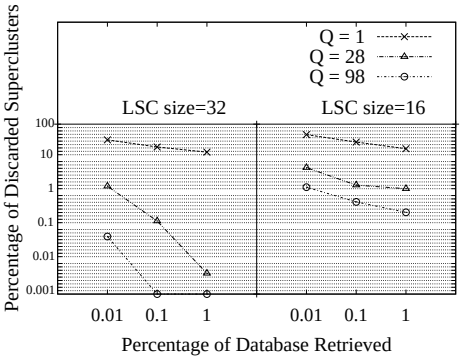
As in all the previous experiments, the kernels are launched with one CUDA Block per query, and each CUDA Block processes a different query.

First, we compare the efficiency of the *LSC* and *LC* indexes in sequential computation. Figure 5.13(a) compares the average of distance evaluations between both indexes, where the *LC* always takes advantage. To understand why the *LSC* is less efficient discarding elements, the Figure 5.13(b) shows the mean of the percentage of discarded superclusters, processing the queries in batches of different sizes (represented by Q in the graph). In this latter figure a supercluster is considered discarded just if its covering radius does not intersect any of the Q queries of the current query batch. Therefore, the larger the size of Q, the less the probability of discarding a supercluster. We observe (in Figure 5.13(b)) that when we try to discard superclusters in query batches of size 98 or higher, the *LSC* is able to discard less than 2% of the superclusters. Figure 5.13(a) represents values with Q=1, and we can observe in the Figure 5.13(b) that the *LSC* is able to discard 69% of the superclusters for Q=1 with the database of 500,000 elements. But even in this scenario, *LC* outperforms *LSC* in the quantity of distance evaluations.

The advantage of the *LC* index is explained because in the case of the *LSC*, each *LC* inside a supercluster is created with a reduced number of elements and the



(a) Average of the distance evaluations (D.E.) per query, between *LSC* of 16 and 32 clusters per supercluster and the *LC* index, using the CoPhIR database.



(b) Discarding of supercluster in the *LSC* with 16 and 32 clusters per supercluster, using the CoPhIR database of 500,000 elements.

Figure 5.13: Results in sequential computation of *LSC* and *LC*.

elements are very close between them, Thus, the centers of clusters and covering radii are of bad quality, and the discard of clusters is low. It is noteworthy that the creation procedure of the *LC* is different of the *LSC*, and therefore their clusters are different (cover different areas, elements and radii), because (as explained in Section 5.2.1) in the *LSC* we first create the superclusters, and after that we create a *LC* index with the local elements of each supercluster.

Despite the total number of distance evaluations increases, we show below that our implementation of *LSC* in GPU outperforms the *LC* one. This counterintuitive behavior is largely explained due to the higher transfer efficiency of the *LSC*. The minimum unit of discarding in the *LC* is a cluster while in the *LSC* is a supercluster, which also is the minimum unit of transfer. Therefore, the layout of the data to be transferred from CPU to GPU gets much more irregular when using *LC*, and the available bandwidth is poorly exploited.

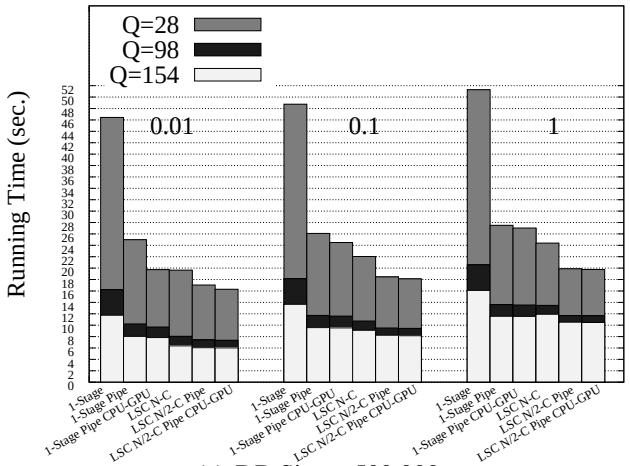
If we think in the clusters as unit of transfers to device memory in the GPU, the *LSC* makes better use of the bandwidth, because each non-discarded supercluster is

a set of clusters that must be processed.

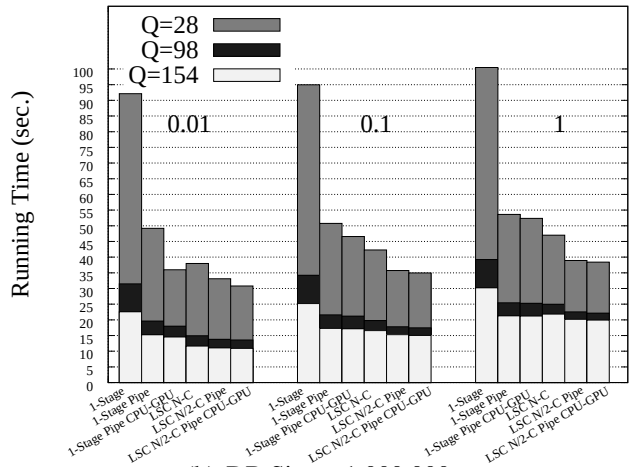
Now, turning our attention to the experiments on the multi-GPU platform, the Figures 5.14(a), 5.14(b) and 5.14(c) present the accumulated running time of the *LC* and *LSC* indexes combined with the pipeline strategies, processing the queries in batches of Q=28, 98 and 154. The different colors of each column represent the running time with a different query batch, for example, in the first column of Figure 5.14(a), the running time of the *1-Stage* strategy, processing the queries in batches of Q=154 is 11.7 seconds, with Q=98 is 16.2 seconds, and with Q=28 is 46.4 seconds. We process the queries in batches of 28, 98 and 154, because these numbers are multiples of 14, which is the number of multiprocessors in our GPUs, and taking account that we are processing each query with a different CUDA Block, a multiple of 14 improves the load balance of CUDA Blocks across multiprocessors.

The former three columns of Figures 5.14(a), 5.14(b) and 5.14(c) are versions of the *LC* with the different pipelines, and the latter three columns are versions of the *LSC* combined with the pipelines. In the following we describe each column. The first column was taken as baseline, and stands for the *1-Stage* strategy (Section 5.1.2). This uses the *LC* index, and after loading N clusters, a kernel is launched to search on them (N is the number of clusters allowed in device memory). The second column (*1-Stage Pipe*) stands for the *1-Stage* strategy, but using the copy-kernel pipeline described in Section 5.2.3, therefore after loading $N/2$ clusters in device memory we launch a kernel to search on them. The third column (*1-Stage Pipe CPU-GPU*) is similar to the second one, but also implements the pipeline CPU-GPU (Section 5.2.2), where the threads that run on CPU-cores try to discard clusters of the *LC* in parallel with the GPUs processing of the previous query batch. The fourth column (*LSC N-C*) stands for the *LSC* index (Section 5.2.1), with N

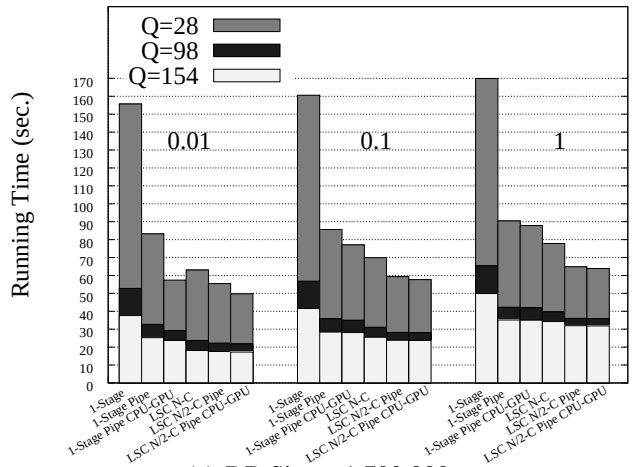
clusters per supercluster, and after loading a supercluster in device memory, a kernel is launched to search on it. The fifth column (*LSC N/2-C Pipe*) stands for the *LSC* index with $N/2$ clusters per supercluster, and using the copy-kernel pipeline, therefore after loading a supercluster by a CUDA stream, a kernel is launched to search on it using the same stream. The last column (*LSC N/2-C Pipe CPU-GPU*) stands for the *Multi-pipeline* strategy described in Section 5.2.4, which uses the *LSC* index with both pipelines.



(a) DB Size = 500,000



(b) DB Size = 1,000,000



(c) DB Size = 1,700,000

Figure 5.14: Running time of the *LC* and *LSC* indexes combined with the asynchronous copies and CPU-GPU pipelines.

In all our experiments we always set the cluster size equal to 256, because we (empirically) found it to be a good parameter. Therefore each supercluster of the *LSC* index is composed by clusters of size 256. We set the clusters allowed in device memory in N=32, and we just copy the results from GPU when a query batch is completely processed. In all the strategies, we copy a cluster of the *LC* (or supercluster in case of *LSC*) with just one `cudaMemcpy`, or one `cudaMemcpyAsync` in the version that implement the copy-kernel pipeline (labeled as *Pipe* in Figure 5.14). All the versions that implement the copy-kernel pipeline require to use page-locked (pinned) memory to transfer data, which increase the bandwidth between host memory and device memory. Because the latter, for fair comparison we used page-locked memory with all the strategies.

Our baseline implementation, labeled as *1-Stage* strategy achieves the worst performance in all the databases for all Q. The *1-Stage Pipe* strategy outperforms the previous one, because it is able to exploit better the copy and kernel engines of the GPU, using them all the time by the pipeline. Therefore, we can hide latencies in the copy instructions and reduce the running time of the search algorithm.The *1-Stage Pipe CPU-GPU* strategy outperforms the previous two, because the reduction in the quantity of clusters copied to device memory, and also because we execute that discard while the GPUs are processing another set of clusters. The *LSC N-C* strategy, despite of performing more distance evaluations than the simpler *LC*, performs better running time because its more efficient management of the bandwidth in the GPU. As we explained in Section 5.2.1, we use one copy instruction to transfer one cluster or one supercluster, because they are the transfer units of the *LC* and *LSC* repectively. But, a supercluster is larger than a cluster, and the fact of transferring larger quantities of data in each copy instruction give the

advantage to the *LSC*. The *LSC N/2-C Pipe* strategy achieves better performance than the previous ones, because the implementation of the copy-kernel pipeline hiding latencies using CUDA streams. Finally, the strategy labeled as *LSC N/2-C Pipe CPU-GPU*, which combines the *LSC* with both pipelines achieves the best performance. The advantages of using the CPU-GPU pipeline in the *LSC* is more evident with Q=28, because the larger Q, the less is the discard of clusters (Figure 5.13(b)). This seems to indicate a certain degree of locality when we process small query batches, but it is lost when the batch is made too large. However, the much larger number of transfers due to a reduced Q does mitigate the benefits of this locality.

To complete our study, we consider the case where the whole database actually fits in the GPUs main memory (i.e. the whole database must just be copied once at the beginning of the process). Our smallest database may be distributed among the two available GPUs, so we could compare our best implementation with this unrealistic scenario. Figure 5.15 shows the results.

Not surprisingly, the *all-fit* implementation (named as *1-Stage DB in Memory* in the figure) outperforms our proposal when searching with small radius. Also as expected, in this ideal version, there is almost no penalty when reducing the Q: it does not entail further transfers, so there is no huge penalty from that side. However, it is very noticeable that, for the largest search radius (1% of the database retrieved) our implementation actually outperforms the *all-fit* version for Q=154. With such a large radius, the discard efficiency is quite low. Thus, kernel execution times are always able to completely hide transfers penalties. Then, we only pay the latency of transferring one *supercluster* per batch of queries. The higher the Q, the less the number of batches and, for this example, we are able to be competitive with

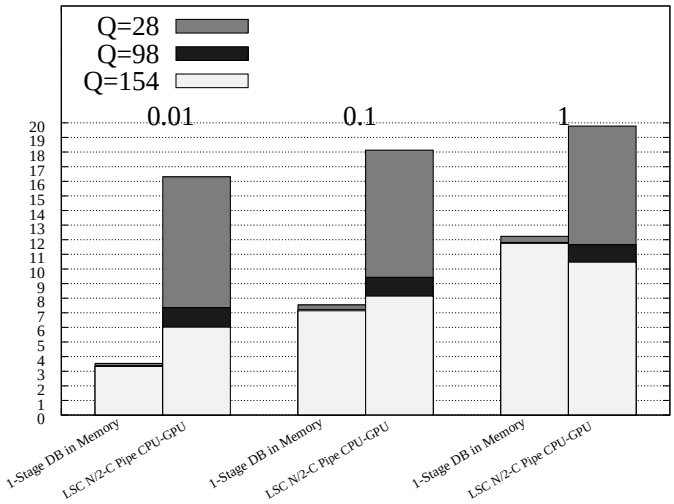


Figure 5.15: *1-Stage* strategy loading all the data in device memory against the *multi-pipeline* strategy with reduced memory size, using the database of 500,000 elements

the *all-fit* version. For the smallest Q value, the number of non-hidden transfers increase too much (and the kernel work decreases), largely degrading performance.

5.3. Conclusions

In this chapter we have presented efficient strategies of distribution and searching to process similarity queries on a multi-GPU platform. We divided the chapter in two sections with different initial assumptions: (1) the first case assume that the whole database fits in device memory, and (2) the second one assumes that just a portion of the database fits in device memory.

In the first section, when the database fits in device memory, we proposed and compared different strategies for the *List of Cluster* (*LC*), exposing the difficulties of dealing with this kind of environment. We obtained a super-linear speed-up

over the single-GPU version, thanks to the *occupancy* (measure of active threads per multiprocessor). When we used the proposed strategy called *1-Stage*, we were able to reduce the quantity of shared memory when we increased the quantity of GPUs, achieving a better occupancy. Also, we validate our proposals in the context of real-time systems, when it is not acceptable to wait for thousands of queries to fill the system before processing them all in parallel, concluding that GPUs can be used for *on-line* query processing in metric spaces as a low-cost high performance alternative to traditional multi CPU implementations.

In the second section, when the database does not fit in memory, we proposed a hierarchical multi-level structure, built on the *LC* index named *List of Superclusters* (*LSC*). The *LSC*, which is composed by *superclusters*, has been designed to perform well on GPUs. A supercluster is made by a center, a covering radius and elements, but with the elements of each supercluster is created a *LC* index into it. Grouping clusters in superclusters allows for a fast discard at CPU level and, using it as the minimal CPU-GPU transfer unit, ensures that the bandwidth is always efficiently exploited. With the objective of dealing with data transfer to (and from) device memory, we implemented a hybrid pipeline CPU-GPU. The CPUs perform a first round of discards for a query batch Q_i while the GPUs are finishing the processing of the previous batch Q_{i-1}. Moreover, the CPU-GPU transfers and the GPU kernels execution is also pipelined using *CUDA streams* and asynchronous copies. The transfer latency is almost completely hidden in that way; indeed, even if the complete list of clusters is copied for each batch query (except those clusters discarded by the CPU), the total exposed latency may be even lower than the experienced when transferring the complete database just once.

Finally, to the best of our knowledge, there is not a public real query log of

similarity search of images. We have presented and made public the first one. Our study with a real query log for similarity search in images, shows that there exits a locality among queries, i.e. the sets of clusters accessed by two consecutive queries have a non null intersection. This could motivate further exploration to reduce transfers by carefully scheduling queries.

Chapter 6

Conclusions and Future Work

This thesis has proposed a set of algorithms and strategies to solve similarity searches in metric spaces using different parallel platforms.

In the first part of the thesis, we have used a multi-core platform, where we found that particular strategies are more suitable depending on the traffic query, obtaining a high speed-up (up to 7.9x with 8 cores) over the sequential algorithm. In the second part, we have used a NVIDIA GPU (Graphic Process Units) graphic card, where we proposed and mapped a set of indexing and exhaustive search strategies to process similarity queries, efficiently exploiting the memory hierarchy of the GPU. We largely outperformed the multi-core version, and we achieved up to 466x of speed-up over the sequential brute force algorithm, solving range queries. In the third part of the thesis we have used a multi-GPU platform, where we extended our previous single-GPU algorithms. We considered two different scenarios: in the simplest one, we assumed that the whole database fits into GPU memory, and in the more realistic scenario we assumed that the database is large enough not to fit in GPU memory.

In the following we give the main conclusions for each parallel platform.

6.1. Multi-core Environment

We used a set of representative and frequently used indexes in the technique literature to show the generality of our proposed algorithms. These algorithms implemented asynchronous multi-thread processing (*Local* strategy) and bulk-synchronous processing (*Bulk-Circular*, *Bulk-Local* and *Bulk-Critical* strategies), where the latter is an implementation of the BSP model.

The *Bulk-Critical* strategy showed the lowest performance, mainly due to the high number of accesses to critical regions (of OpenMP), and the high cost each access implies. The code of a critical region is sequentially executed by just one thread at a time, thus its high access decrease the performance.

The *Local* strategy showed the highest performance under a high query traffic situation, achieving up to 7.8x of speed-up with 8 cores, over the sequential version. In this strategy each thread process its queries completely, without communication with others threads, and avoiding synchronizations between them.

The *Bulk-Circular* strategy showed the highest performance under a low query traffic situation (up to 7.9x of speed-up). In this strategy each thread distributes the *tasks* of its queries among all the other threads, which means that all the threads cooperate to solve every query. This takes advantage under a low query traffic mainly because reduces the idle time of the threads.

The *Bulk-Circular* strategy showed the highest performance under a low query traffic situation. In this strategy each thread distributes the *tasks* of its queries among all the other threads, which means that all the threads participate in the solution of all

the queries. This takes advantage under a low query traffic mainly because reduces the idle time of the threads.

According to the previous results, we proposed a *hybrid* strategy, which is able to change between the *Local* and *Bulk-Circular* strategies, depending on the current query traffic. This strategy showed the best performance in a scenario where the traffic can change, because it is able to exploit the advantages of both, *Local* and *Bulk-Circular* strategies.

Also, we compared two different distributions of the database. The first, distributes the elements of the database among the threads, and each thread creates its own index. The second, keep just one global index in memory. The latter showed the best performance regarding to the quantity of distance evaluations and running time. This is due to the quality that show the global centers (or pivots), increasing the percentage of discarded elements.

6.2. Single-GPU Environment

We proposed and compared different algorithms of brute force and indexing for the two main kind of queries (*range* and *k*NN queries). We used the metric indexes *List of Cluster* (*LC*) and *SSS-Index* because: (1) their good results previously observed in a multi-core platform, and (2) work on dense matrices, and (3) exhibit certain regularity in the access pattern. All these features are very suitable to improve coalescing of memory access in the GPU. In our exploration we have found that some optimum parameters in GPU are very different of those used in sequential computing, in particular with the *SSS-Index* the optimum is reached with just one pivot for vector databases.

Due to the complexity and restrictions of the GPU, we found different problems for both kinds of queries, *range* and *k*NN queries, thus we applied different parallelization strategies for each type.

With regarding to our proposals to solve *k*NN queries, they used a set of heaps stored in device memory to keep the K nearest elements to the query across the search process. Afterwards, a *warp* and the first thread of the CUDA Block reduce the elements of the heaps in just one heap stored in shared memory with the final results of the query. We outperformed the previous methods of the related work based on sorting. Also, we showed that the *increasing range* method is more suitable on a GPU environment than the decreasing one, which is opposite to what occurs in sequential computing.

In both kind of queries, the *LC* index reached the best performance given its good regularity and its access pattern to the device memory. It achieves up to 466x of speed-up over the sequential brute force algorithm.

6.3. Multi-GPU Environment

We divided this part in two sections with different initial assumptions: (1) the first case assumes that the whole database fits in GPU memory, and (2) the second one assumes that just a portion of the database fits in GPU memory.

In the first section, when the database fits in GPU memory, we proposed and compared different strategies for the *LC* index, exposing the difficulties of dealing with this kind of environment. We obtained a super-linear speed-up over the single-GPU version, thanks to the *occupancy* (measure of active threads per multiprocessor). When we used the proposed strategy called *1-Stage*, we were able to reduce

the quantity of shared memory when we increased the quantity of GPUs, achieving a better occupancy. Also, we validated our proposals in the context of real-time systems, when it is not acceptable to wait for thousands of queries to fill the system before processing them all in parallel, concluding that GPUs can be used for *on-line* query processing in metric spaces as a low-cost high performance alternative to traditional multi CPU implementations.

In the second section, when the database does not fit in GPU memory, we proposed a hierarchical multi-level structure, built on the *LC* index named *List of Superclusters* (*LSC*). The *LSC*, which is composed by *superclusters*, has been designed to perform well on GPUs. A supercluster is made by a center, a covering radius and elements, but with the elements of each supercluster is created a *LC* index into it. Grouping clusters in superclusters allows for a fast discard at CPU level and, using it as the minimal CPU-GPU transfer unit, ensures that the bandwidth is always efficiently exploited. With the objective of dealing with data transfer to (and from) GPU memory, we implemented a hybrid pipeline CPU-GPU. The CPUs perform a first round of discards for a query batch Q_i while the GPUs are finishing the processing of the previous batch Q_{i-1}. Moreover, the CPU-GPU transfers and the GPU kernel executions are also pipelined using *CUDA streams* and asynchronous copies. The transfer latency is almost completely hidden in that way; indeed, even if the complete list of clusters is copied for each batch query, the total exposed latency may be even lower than the experienced when transferring the complete database just once. The query log used in this section is particular, because to the best of our knowledge, there is not a public real query log of similarity search of images. But, we have presented and made public the first one. Our study with a real query log for similarity search in images, shows that there exits a locality among queries, i.e. the

sets of clusters accessed by two consecutive queries have a non null intersection. This could motivate further exploration to reduce transfers by carefully scheduling queries.

6.4. Future Work

Several proposals remain for the future development, in the following we give some of them:

- To extend our algorithms for being used over a distributed memory system parallel platform.
- To analyze and evaluate the performance of metric structures based on tree, which could efficiently exploit the memory hierarchy of the GPU.
- To propose algorithms that efficiently exploit the rest of the memory hierarchy.
- To analyze different methods in GPU to perform efficient searches in high dimensional spaces.
- To extend the proposed algorithms to implement dynamism in the data structure used in GPU.
- To evaluate the impact of using different tools of programming from CUDA, such as OpenCL, OpenACC, and others based on new programming models as MPI/OmpSs.
- To extend the proposed algorithms in GPU to be used in other heterogeneous platforms, for example using FPGAs.

- To exploit the locality of consecutive queries implementing cache strategies, using a real query log.

Apéndice A

Resumen en Español

En cumplimiento del Artículo 4 de la normativa de la Universidad Complutense de Madrid que regula los estudios universitarios oficiales de postgrado, se presenta a continuación un resumen en español de la presente tesis que incluye la introducción, objetivos, principales aportaciones y conclusiones del trabajo realizado.

A.1. Introducción

En bases de datos tradicionales, el procesamiento de una consulta devuelve resultados que son *exactamente iguales* a la consulta dada. Pero, con la evolución de las tecnologías de información y comunicación, han emergido nuevos repositorios de información con tipos de datos no tradicionales. Tipos de datos, tales como audio, video o imágenes, que no pueden ser estructurados de una forma tradicional bajo tuplas o llaves, pero actualmente hay un creciente interés por realizar búsquedas en este tipo de información. Por lo tanto, se hace necesaria la creación de nuevos

modelos para búsqueda en repositorios no estructurados, y donde la búsqueda de resultados *exactamente iguales* a la consulta carece de sentido.

El primer concepto a tener en cuenta para poder encontrar una solución, es el de *búsqueda por similitud* [60], es decir, búsqueda de los elementos de la base de datos que son similares o cercanos a la consulta dada. La similitud es medida con una función de distancia que satisface la propiedad de desigualdad triangular y el conjunto de objetos es llamado *espacio métrico*.

Una técnica muy difundida en los últimos años para indexar y buscar eficientemente objetos complejos son los llamados *índices* para espacios métricos. Se han propuesto numerosas estructuras de datos para computación secuencial basada en esta técnica, que pueden alcanzar buena eficiencia comparado con búsquedas en espacios multi-dimensionales que contienen gran número de objetos. No obstante, el diseño de estos índices ha sido orientado a la optimización de consultas individuales en computación secuencial, resolviendo los dos principales tipos de consulta por similitud: consultas por *rango* y consultas *k*NN. Una consulta por rango, representada como (q, r), es la operación que recupera de la base de datos el conjunto de objetos cuya distancia a la consulta q no es mayor que r. Una consulta *k*NN (*k* nearest neighbors), representada como *k*NN(q) recupera de la base de datos los k elementos más cercanos a q.

Debido a que el problema ha aparecido en diversas áreas, las soluciones han provenido de campos tales como estadísticas, geometría computacional, inteligencia artificial, bases de datos, bio-informática, reconocimiento de patrones, minería de datos, la Web. Actualmente los buscadores para la Web indexan docenas de billones de documentos y cientos de millones de otros tipos de objetos complejos tales como datos multimedia. Por ejemplo, recientemente ha aparecido el primer

buscador comercial (*Google Goggles* [2]), que permite entregar una imagen como consulta, y aunque sólo presenta buen funcionamiento con ciertos objetos, es el principio de este tipo de aplicaciones.

Las cargas de trabajo en los grandes buscadores se caracterizan por la existencia de una gran cantidad de consultas siendo procesadas en todo momento sobre un conjunto muy grande de objetos (cientos de millones). En estos sistemas la métrica de interés a ser optimizada es el *throughput*, que se define como la cantidad de consultas completamente resueltas por unidad de tiempo. Para alcanzar altas tasas de respuesta sobre cientos de millones de objetos con miles de consultas por segundo, es necesario utilizar técnicas de computación paralela. En este caso la paralelización se realiza sobre decenas o cientos de *nodos* (procesadores) sobre los cuales se distribuyen uniformemente los objetos e índices, y donde cada nodo puede contener varios CPU-cores y GPUs. La contribución principal de esta tesis está enfocada en la búsqueda eficiente en espacios métricos sobre uno de los nodos antes mencionados, utilizando un entorno de memoria compartida.

Para el presente trabajo se utilizaron como base índices métricos que ya existían en la literatura, que son capaces de utilizar algoritmos secuenciales para resolver consultas en espacios métricos de forma eficiente. Como se mencionó anteriormente, estos índices han sido optimizados para resolver consultas individuales, y no para la resolución de un conjunto de ellas en paralelo, ni para paralelizar la resolución de una de ellas. La primera parte de esta tesis propone estrategias de distribución y búsqueda para resolver consultas por similitud en espacios métricos, utilizando un servidor multi-core bajo un sistema de memoria compartida.

Durante los últimos años, ha aparecido una alternativa muy prometedora para la aceleración de procesos de búsqueda, son tarjetas gráficas creadas por NVIDIA

denominadas GPU (Graphics Processing Units). Las consultas por rango y kNN proveen diferentes niveles de paralelismo: se puede procesar un conjunto de consultas en paralelo, un conjunto de evaluaciones de distancia en paralelo para una consulta dada, o incluso explotar paralelismo en la operación de distancia misma. Este esquema se adapta muy bien a la arquitectura de la GPU, la que implementa una organización de los hilos a varios niveles. Estas arquitecturas poseen una compleja jerarquía de memoria, en donde algunas de ellas pueden ser controladas mediante software. Estudios empíricos muestran que esto último es crucial para explotar eficientemente este sistema de memoria, y para lograr una mejora significativa cuando se utiliza una GPU para la aceleración de aplicaciones [48]. En la segunda parte de la presente tesis, se proponen estrategias para mapear y acelerar el proceso de búsqueda utilizando una tarjeta gráfica GPU, y en la tercera parte de esta tesis se extienden los algoritmos previos a una plataforma híbrida multi-core y multi-GPU.

También se estudió el caso de las bases de datos suficientemente grandes para no caber en la memoria de la GPU. Más específicamente, se implementó un algoritmo híbrido que hace uso de los cores en CPU GPU. También se presenta un nuevo índice denominado *Lista de Superclusters* (*LSC*), que presenta propiedades convenientes para la transferencia de memoria en GPU.

A.2. Conocimiento Previo

A.2.1. Espacios Métricos y Búsquedas por Similitud

Un *espacio métrico* es un conjunto X de objetos válidos, con una función de distancia $d : X^2 \to \mathbb{R}$, tal que $\forall x, y, z \in X$ cumple con las siguientes propiedades:

- Positividad : $d(x,y) \geq 0$, $x \neq y \Rightarrow d(x,y) > 0$.
- Simetría: $d(x,y) = d(y,x)$.
- Desigualdad triangular : $d(x,y) + d(y,z) \geq d(x,z)$.

Entonces, el par (X,d) es llamado *Espacio Métrico*.

Sea $Y \subseteq X$ el conjunto de objetos que componen la base de datos. El concepto de *búsqueda por similitud* consiste en recuperar todos los objetos pertenecientes a Y que sean parecidos a un elemento de consulta q que pertenece al espacio X.

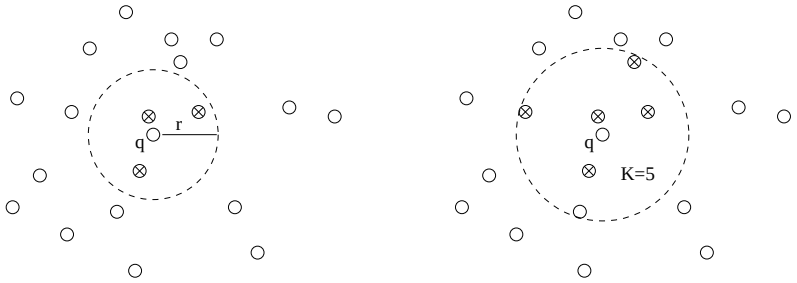
Las consultas por similitud sobre espacios métricos son básicamente dos:

1. **Consulta por Rango** (q,r) **[17]:** Sea un espacio métrico (X,d), un conjunto de datos finito $Y \subseteq X$, una consulta $q \in X$, y un radio $r \in \mathbb{R}$. La consulta por rango q con radio r es el conjunto de puntos $y \in Y$, tal que $d(q,y) \leq r$.

2. **Los** k **Vecinos más Cercanos** k**NN(**q**) [21]:** Sea un espacio métrico (X,d), un conjunto de datos finito $Y \subseteq X$, una consulta $q \in X$ y un entero k. Los k vecinos más cercanos a q son un subconjunto A de objetos de Y, donde la $|A| = k$ y no existe un objeto $y \in X - A$ tal que $d(y,q)$ sea menor a la distancia de algún objeto de A a q.

En la Figura A.1 se ilustran ambos tipos de consulta. Para mayor claridad las consultas están realizadas sobre un conjunto de puntos en $\mathbb{R}^2$(espacio métrico). A la izquierda se muestra una consulta por rango con radio r y a la derecha una consulta de los 5-vecinos más cercanos a q. En este último caso, también se gráfica el radio necesario para encerrar los 5 puntos. Se puede observar que dada una consulta q y una cantidad k (en este ejemplo 5), es posible que existan distintas respuestas.

En [17] se muestran dos métodos principales para resolver consultas de tipo kNN. A continuación se describen ambos métodos.



(a) Consulta por rango (q,r) (b) Consulta kNN(q)

Figura A.1: Ejemplos de consultas.

1. **Radio Creciente:** Este algoritmo de búsqueda de los k vecinos más cercanos está basado en un algoritmo de búsqueda por rango de la siguiente forma: buscar q con radio fijo r=$a^i\varepsilon$ ($a > 1$, $\varepsilon \in codom(d)$), con $i = 0$ al comienzo e incrementarlo hasta que al menos k elementos son abarcados con $r = a^i\varepsilon$. Luego, el radio es ajustado entre $r = a^{i-1}\varepsilon$ y $r = a^i\varepsilon$ hasta que k elementos son alcanzados.

2. **Radio Decreciente:** Este algoritmo de búsqueda comienza con una búsqueda por rango (r) con $r = \infty$, y una vez que se han alcanzado k elementos, el radio es ajustado a $r \leftarrow min(r, d(q,e))$ (siendo e un nuevo elemento con el que se debe comparar la consulta q), con la ayuda de una cola de prioridad.

A.2.2. Indexación

Teniendo en cuenta que la función de distancia es *computacionalmente costosa* de calcular, la indexación surge como una alternativa a la solución trivial de búsqueda exhaustiva (que toma $O(n)$ para una base de datos con n elementos).

Por lo tanto, se hace necesario preprocesar la base de datos, para lo cual se paga un costo inicial de construcción de un *índice* a fin de ahorrar cálculos de distancia al momento de resolver las búsquedas. Esto último se realiza descartando

objetos utilizando la propiedad de desigualdad triangular. En muchas aplicaciones la evaluación de la distancia es tan costosa que los demás factores pueden ser despreciadas.

Todos los algoritmos de indexación particionan la base de datos en subconjuntos. Para particionar la base de datos existen dos grandes enfoques: *algoritmos basados en pivotes* y *algoritmos basados en clustering o particiones compactas* [17].

Los algoritmos basados en pivotes realizan una preselección de objetos de la base de datos. Estos objetos (o pivotes), se utilizan para descartar objetos usando la propiedad de desigualdad triangular. Algunos ejemplos son *SSS-Index* [8], *FQT* y sus variantes [4], *Spaghettis* y sus variantes [14, 42].

Los algoritmos basados en clustering dividen el espacio en *áreas*, donde cada área tiene un *centro* o *split*. Se almacena alguna información sobre el área que permita descartarla completamente realizando tan sólo una comparación entre la consulta con su centro. Algunos ejemplos son *GNAT* [7], *EGNAT* [41], *M-tree* [18] y *Lista de Clusters* [16].

En el presente trabajo se seleccionaron y realizaron experimentos utilizando 5 índices distintos, debido a que éstos son ampliamente citados en la literatura técnica y porque todos poseen diferentes características abarcando un gran abanico de modelos de búsqueda. Estos son el *EGNAT* [41], *M-tree* [18], *SSS-Index* [8], *SSS-Tree* [9] y la *Lista de Clusters* (*LC*) [16].

En particular, el SSS-Index y LC también fueron seleccionados para ser utilizados sobre la GPU, debido a que las características de su estructura son favorables para la tarjeta gráfica.

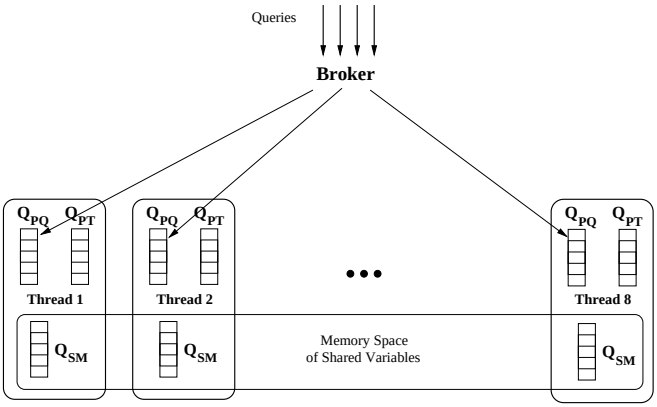


Figura A.2: Modelo de Búsqueda usado en los experimentos.

A.3. Estrategias de Distribución y Búsqueda sobre una Plataforma multi-core

La arquitectura utilizada para realizar nuestras propuestas sobre una plataforma multi-core está representado por la Figura A.2, en donde las consultas entrantes son distribuidas por una máquina *broker*. Las consultas se distribuyen de forma circular entre los threads.

Cada thread posee tres colas: (1) la *Cola Privada de Consultas* (Q_{PC}), que mantiene las consultas suministradas por el broker; (2) la *Cola Privada de Requerimientos* (Q_{PR}), en donde están los *requerimientos* que deben ser procesados, y donde un *requerimiento* es una estructura que indica una cierta cantidad de evaluaciones de distancias a ser ejecutadas por un thread durante la solución de una consulta; (3) la *Cola Secundaria de Mensajes* (Q_{SM}), donde se encuentran los requerimientos destinados a otros threads.

Las colas Q_{PC} y Q_{PR} son privadas a cada thread. En cambio, la cola Q_{SM} es global y todos los threads tienen acceso a ella, para permitir el paso de mensajes

(como se explica en la Sección A.3.3). Debido a la variación que puede sufrir el tamaño de la cola Q_{PR}, sus elementos son creados dinámicamente.

A.3.1. Descripción de la Búsqueda

La búsqueda está basada en el procesamiento de *requerimientos*. Un requerimiento es una estructura formada por los siguientes campos:

1. **Región:** indica la región del índice que se debe acceder para procesar la consulta. En el caso de los índices basados en árboles este campo es un puntero a un nodo.

2. **Indice:** indica el elemento del nodo donde se debe comenzar a buscar. Esto es usado para retomar una búsqueda interrumpida por haber alcanzado R evaluaciones de distancia.

3. **Consulta:** la consulta misma.

4. **Campo extra:** datos específicos requeridos por la búsqueda sobre el índice.

De esta forma cada requerimiento implica realizar L evaluaciones de distancia, donde $1 <= L <= N_{region}$ (N_{region} es la cantidad de elementos almacenados en la zona de memoria apuntada por el campo *Región*). En el caso de los índices basados en árboles N_{region} es la cantidad de elementos de un nodo.

Según lo anterior, cuando una consulta se obtiene desde la Q_{PC}, se genera un *requerimiento inicial*, que es agregado a la Q_{PR} para comenzar su búsqueda. Dependiendo del índice, procesar un requerimiento puede generar más requerimientos, que al ser procesados generarán aún más, y así sucesivamente hasta completar todos los requerimientos que exige la consulta.

A continuación se describen las estrategias multi-core implementadas. En todas ellas no se realiza particionado del índice, es decir, se mantiene sólo un índice global en memoria al que acceden todos los threads.

A.3.2. Estrategia Local

En esta estrategia los threads obtienen una consulta desde Q_{PC}, de donde se obtiene el requerimiento inicial, y éste se agrega a Q_{PR}. Todos los requerimientos generados se almacenan en la misma Q_{PR}, que es local y privada a cada thread. La cola Q_{SM} no se usa en esta estrategia.

Lo anterior implica que cada thread resuelve cada consulta completamente y de forma aislada. Es decir, cada thread es capaz de resolver sus consultas de forma incomunicada del resto de los threads, evitando instrucciones de sincronización y de paso de mensajes, pero comprometiendo el balance de carga.

A.3.3. Estrategia Bulk-Circular

Esta estrategia procesa consultas utilizando el modelo BSP (Bulk Synchronous Parallel) [57], el que define un comportamiento sincrónico entre los threads involucrados. Para implementar el modelo BSP, y poder crear una secuencia de pasos (denominados *supersteps*) se utilizó la directiva de OpenMP *#pragma omp barrier*.

En esta estrategia cada thread usa las tres colas Q_{PC}, Q_{PR} y Q_{SM}. Cada thread almacena en su propia Q_{SM} los *mensajes* que son destinados a los demás threads. Un *mensaje* es un requerimiento más el identificador del thread destino.

Lo anterior implica que los requerimientos generados no se almacenan solamente en Q_{PR} como en la estrategia Local, sino que los requerimientos destinados

a otros threads se almacenan en forma de mensaje en Q_{SM}. Cada thread selecciona el destino de un mensaje siguiendo una distribución circular.

El algoritmo que describe el proceso de búsqueda para esta estrategia establece dos *supersteps*. El primero inicializa la cola Q_{SM} borrando todos sus elementos, luego se obtienen requerimientos de la Q_{PR} y se procesan. A cada nuevo requerimiento generado se le asigna (circularmente) un thread de destino. Si el destino es el thread actual, entonces se almacena el requerimiento en Q_{PR}, de lo contrario, se almacena en Q_{SM} con el identificador del thread destino correspondiente. Este primer paso se realiza hasta completar un máximo de R evaluaciones de distancia (si Q_{PR} está vacía entonces se obtiene una consulta nueva desde la Q_{PC} y se inserta el requerimiento inicial de la consulta en Q_{PR}). Cuando se alcanzan R evaluaciones de distancia, se aplica una función de sincronización y se continúa con el segundo superstep, donde cada thread lee las Q_{SM} de los demás y rescata los mensajes que están destinados para él. Todos estos mensajes son insertados en forma de requerimiento en la Q_{PR}. Cuando el segundo paso acaba, se aplica nuevamente una función de sincronización, y luego se continúa con el primer superstep, y así sucesivamente. Cabe destacar que la escritura y lectura a las colas Q_{SM} se realizan en diferentes supersteps, lo que implica que no hay problemas de conflictos entre lecturas y escrituras concurrentes.

A.3.4. Estrategia Bulk-Critical

Esta estrategia es muy similar a la Bulk-Circular, pero usa *regiones críticas* de OpenMP para implementar el paso de mensajes. Una región crítica corresponde a una porción de código que se ejecutará por sólo un thread a la vez. Es decir, si un

thread desea ejecutar instrucciones que se encuentran en una región crítica, sólo lo podrá hacer si ningún otro thread se encuentra ejecutando instrucciones de dicha región. En caso contrario, el thread esperará hasta que la región quede disponible.

Cada vez que se lee o escribe un mensaje por cualquier thread, esto se realiza en la misma región crítica, pero la lectura de las Q_{SM} del resto de los threads se realiza sólo después de alcanzar R evaluaciones de distancia.

Esta estrategia, al utilizar regiones críticas, tiene la ventaja de evitar instrucciones de sincronización. Pero posee la desventaja de que el acceso secuencial a las regiones críticas puede actuar como cuello de botella.

A.3.5. Estrategia Bulk-Local

Esta estrategia es similar a la Bulk-Circular pero con la diferencia que el destino es siempre el mismo thread, por lo que no existe Q_{SM}, lo que implica que no hay intercambio de mensajes. Debido a esto último, ésta es una estrategia local donde cada thread resuelve completamente una consulta, pero realiza procesamiento por lotes de forma síncrona.

A.3.6. Resultados Experimentales sobre una Plataforma multi-core

Todos los experimentos fueron realizados en una máquina con 2 CPU's Intel Quad-Xeon de 2.66 GHz, cada una con 4 núcleos, y 16GB de memoria.

Los experimentos se realizaron con 2 bases de datos:

1. ***Words*** **:** Diccionario español con 51589 palabras. Se usó la *distancia de edición* (o *distancia de Levenshtein*) [32] con radio 1, 2 y 3, pues éstos son radios

usados en trabajos previos [41, 35, 40]. Esta distancia entrega la cantidad mínima de inserciones, eliminaciones o reemplazos para que una palabra sea igual a otra. Las consultas para esta base de datos fue un archivo de 40000 consultas, obtenidas desde la Web Chilena en el dominio todo.cl.

2. ***Images*** **:** Esta base de datos fue creada a partir de una colección de 40701 vectores que representan imágenes de la NASA, y éstas se usaron como una distribución de probabilidades para generar vectores aleatorios hasta completar 120,000 imágenes de dimensión 20. Se usó la *distancia euclidiana* para medir la similitud entre los objetos. Los radios utilizados fueron los que permiten recuperar el 0.01 %, 0.1 % y 1 % de la base de datos por consulta. Estos son valores usados en trabajo previos [41, 16, 40]. El 80 % de la base de datos se usó para la construcción del índice y el 20 % restante como archivo de consultas.

Los experimentos fueron normalizados al mayor valor observado, para apreciar mejor las diferencias reportadas por las diferentes estrategias. Se utilizaron escenarios de alto y bajo tráfico de consultas entrantes al sistema.

La Figura A.3 muestra el tiempo de ejecución para las estrategias *Bulk-Circular* y *Bulk-Critical*. El experimento se realizó simulando un alto tráfico de consultas y usando el índice *EGNAT*. *Bulk-Critical* presenta tiempos de ejecución muy altos, debido a que las regiones críticas son muy accedidas, y como es sabido ([13]), esto degrada mucho el rendimiento del programa. Por este motivo esta estrategia fue descartada para los experimentos siguientes. Un comportamiento similar se observó con los demás índices.

La Figura A.4 muestra los experimentos realizados con todas las estrategias bajo una situación de alto tráfico de consultas. En cambio, la Figura A.5, muestra los mismos experimentos pero con un bajo tráfico de consultas.

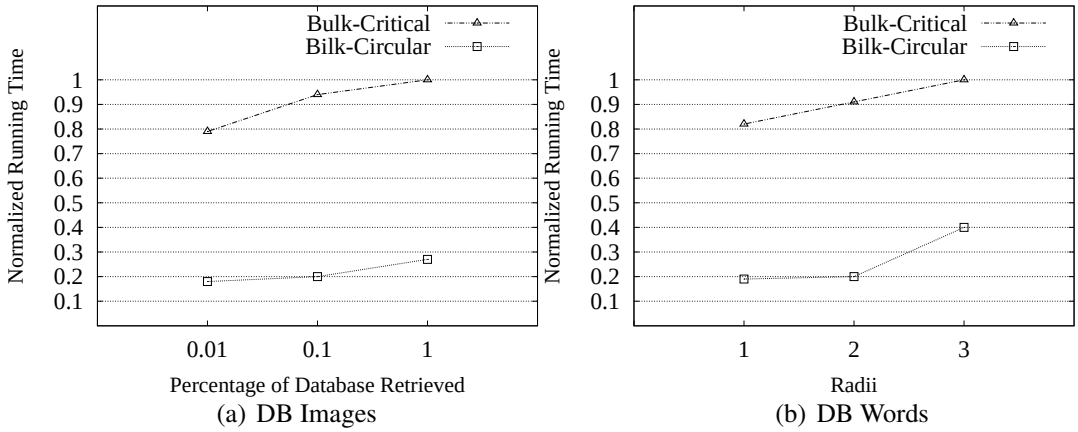


Figura A.3: Tiempo de Ejecución para la estrategias *Bulk-Circular* y *Bulk-Critical*, usando el *EGNAT* con un alto tráfico de consultas.

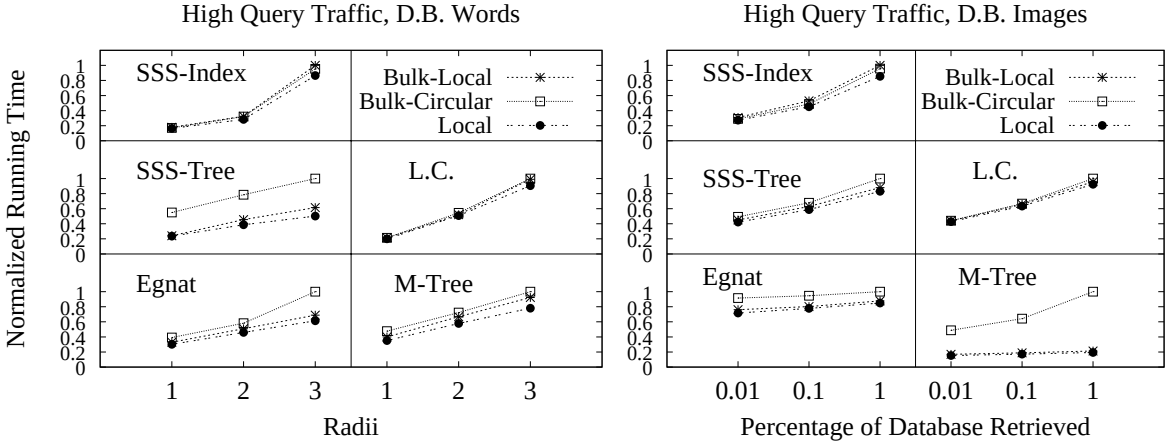


Figura A.4: Tiempo de Ejecución para un alto tráfico de consultas.

Los resultados indican que con un alto tráfico la estrategia *Local* obtiene un mejor rendimiento en todos los índices. Esto es debido al costo que implica la sincronización y el paso de mensajes para el caso de la *Bulk-Circular*, pero esta última toma ventaja con un bajo tráfico de consultas. Esta ventaja se debe principalmente

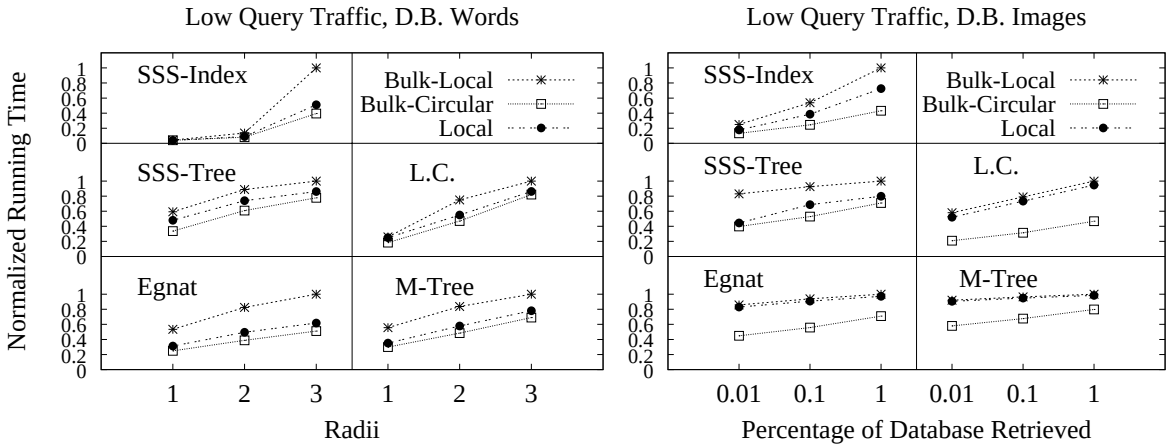


Figura A.5: Tiempo de Ejecución para un bajo tráfico de consultas.

a que la estrategia *Bulk-Circular* reduce los tiempos de ociosidad de los threads que se encuentran en espera de una consulta. Para tráfico alto la estrategia *Bulk-Local* alcanza un rendimiento similar al de la estrategia *Local* para la mayoría de los índices estudiados.

Para ilustrar de mejor manera lo que sucede en un escenario de baja frecuencia de consultas, la Figura A.6 muestra un ejemplo de dos distribuciones de consultas con baja frecuencia, procesadas usando 4 threads. La primera, es cuando el arribo de consultas corresponde a la mejor distribución posible, es decir, las consultas se distribuyen de tal forma que todos los threads harán (aproximadamente) la misma cantidad de evaluaciones de distancia. La segunda, es una distribución desfavorable, donde el primer thread procesa las consultas ($q1$, $q5$, $q9$) que implican más evaluaciones de distancia. En ambos casos la estrategia *Bulk-Circular* obtiene el mayor throughput (consultas resueltas por unidad de tiempo). La frecuencia de arribo de consultas para este ejemplo (Figura A.6) fue: en la primera unidad de tiempo arriban

2 consultas, en la segunda 1 consulta y en la tercera 1 consulta, y luego se repite este ciclo.

Favorable Arrival

Time	Local	Bulk–Circular
	T1 T2 T3 T4	**T1 T2 T3 T4**
1	q1 q2	q1 q1 q1 q1
2	q1 q2 q3	q2 q2 q2
3	q1 q2 q3 q4	q3 q3 q4
4	q1 q5 q6	**q5 q5 q5 q5**
5	q5 q6 q7	q6 q6 q6 q7
6	q8 q5 q6 q7	q7
7	q9 q5q10	q8 q10 q10 q10
8	q9 q10 q11	q9 q9 q9 q9
9	q9q12 q10 q11	q11 q12 q11
10	q9	
11		
12		

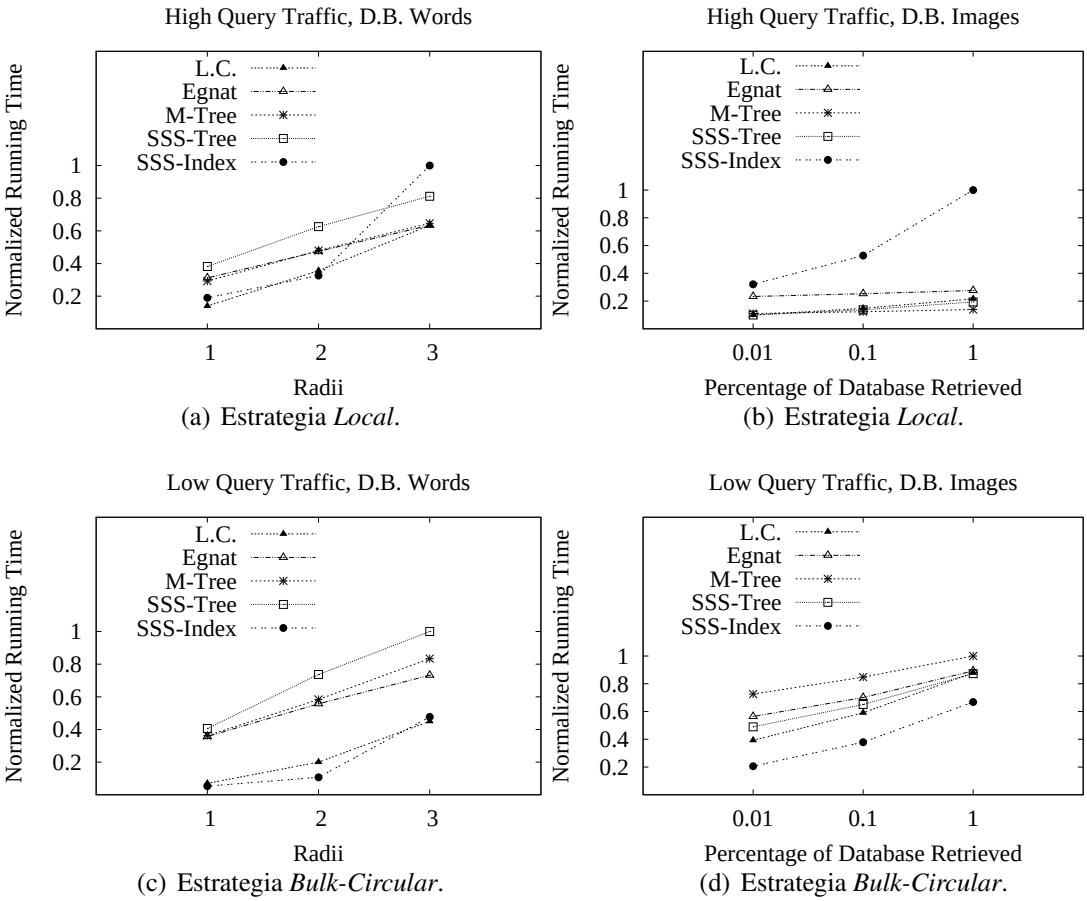
Unfavorable Arrival

Time	Local	Bulk–Circular
	T1 T2 T3 T4	**T1 T2 T3 T4**
1	q1 q2	q1 q1 q1 q1
2	q1 q2 q3	q2 q2 q2
3	q1 q2 q3 q4	q3 q3 q4
4	q1 q6	q5 q5 q5 q5
5	q5 q6 q7	q6 q6 q6
6	q5 q6 q7 q8	q7 q7 q8
7	q5 q10	q9 q9 q9 q9
8	q5 q10 q11	q10 q10 q10
9	**q9** q10 q11 q12	q11 q11q12
10	**q9**	
11	**q9**	
12	**q9**	

Figura A.6: Distribuciones de consultas en un escenario de baja frecuencia.

La Figura A.7 muestra la comparación entre los índices usando la estrategia *Local* para alta frecuencia de consultas y *Bulk-Circular* para baja frecuencia. Esto con la finalidad de observar cuál índice se adapta mejor a la implementación de las estrategias. El índice con un buen desempeño tomando en cuenta ambos escenarios de frecuencia fue la *LC*. Una de las razones de esto es que el requerimiento de este índice implica una cantidad fija de evaluaciones de distancia, ya que éste encierra a un cluster completo, y esto favorece el balance de carga.

La Figura A.8 muestra el speed-up de los índices sobre la base de datos *Words* con radio 3. El speed-up S está dado por $S = T_s/T_m$, donde T_s es el tiempo de ejecución de la aplicación secuencial y T_m el tiempo de la estrategia multi-core. El mayor speed-up, usando alta frecuencia de consultas, lo obtiene la estrategia *Local*, mientras que con baja frecuencia de consultas, *Bulk-Circular* es la mejor estrategia.

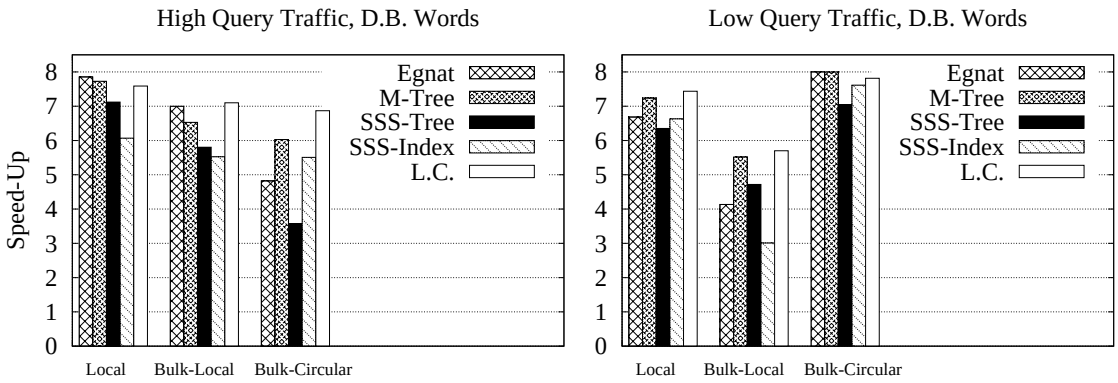


(a) Estrategia *Local*.
(b) Estrategia *Local*.
(c) Estrategia *Bulk-Circular*.
(d) Estrategia *Bulk-Circular*.

Figura A.7: Comparación de los índices con la estrategia *Local* para alto tráfico de consultas y la *Bulk-Circular* para bajo tráfico.

A.3.7. Estrategia híbrida

Esta estrategia es capaz de aplicar un intercambio entre la estrategia *Local* y *Bulk-Circular* dependiendo del tráfico de consultas entrantes. Cuando el número de consultas en espera C_p satisface $C_p > P * C_{max}$ (P: cantidad de threads, C_{max}: Número de consultas que indica el límite entre una situación de tráfico bajo y una de tráfico alto) se comienza a procesar los requerimientos según la estrategia *Local*



(a) Tráfico Alto de Consultas (b) Tráfico Bajo de Consultas

Figura A.8: Speed-up de los índices sobre la base de datos *Words* con radio 3.

y cuando el número de consultas en espera es suficientemente bajo se cambia a la estrategia *Bulk-Circular*

La Figura A.9 muestra el throughput de las consultas en el sistema, es decir, la cantidad de consultas resueltas por unidad de tiempo. En esta figura se muestran las estrategias *Local*, *Bulk-Circular* e *Híbrida* utilizando la *LC*, variando la frecuencia de consultas entrantes. La estrategia *Híbrida* es la que muestra el mayor throughput, pues es la que explota las ventajas de ambas estrategias.

A.4. Estrategias de Distribución y Búsqueda en GPU

En esta sección se proponen y comparan algoritmos de búsqueda de índices métricos sobre GPU (Graphic Processor Unit) basados en CUDA [22]. Los índices seleccionados para ser implementados en GPU fueron la *LC* y *SSS-Index*, debido a que: (1) ambos almacenan su índice en matrices, y además presentan cierta regular-

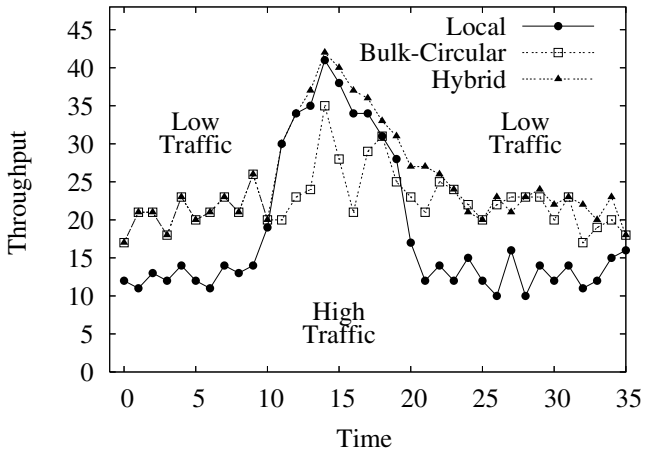


Figura A.9: Throughput: consultas completamente resueltas por unidad de tiempo, usando el índice *LC*.

idad en el acceso a memoria, y como es sabido ([48]), esto favorece al rendimiento en GPU; (2) ambos índices mostraron en la sección anterior muy buenos sobre una plataforma multi-core.

En todas las implementaciones siguientes cada bloque de threads (*CUDA Block*) se encarga de resolver una consulta completamente, pues de esta forma se pueden resolver varias consultas en sólo un lanzamiento de *kernel*, dado que el lanzamiento sucesivo de éstos degrada el rendimiento. Además, de esta forma los threads encargados de resolver una misma consulta pueden usar sincronización, que está habilitada sólo para los threads pertenecientes al mismo bloque, evitando de esta forma conflictos de concurrencia. Es decir, se explota paralelismo a dos niveles: (i) *Paralelismo de grano grueso* al resolver un conjunto de Q consultas en paralelo en sólo un lanzamiento de *kernel*, y (ii) *Paralelismo de grano fino* al resolver cada consulta con un conjunto de threads.

Por tanto, cada *kernel* es lanzado con Q bloques (Q=número de consultas a

resolver) maximizando el número de threads por bloque. Si Q sobrepasa el máximo permitido de bloques, entonces se deben hacer sucesivos lanzamientos de *kernels* hasta resolver todas las consultas. En los experimentos realizados para el presente trabajo, no fue necesario lanzar más de un *kernel*. La consulta que debe resolver el bloque, siempre se almacena en *shared memory* debido al frecuente acceso a ésta por parte de los threads.

Debido a la complejidad y restricciones de la GPU, se analizan los dos tipos de consultas (consultas por rango y consultas *k*NN) de forma separada. Para ambos tipos de consultas, las estrategias empleadas y dificultades encontradas fueron diferentes.

A continuación, en la Sección A.4.1 se describen nuestras propuestas y experimentos para resolver consultas por rango en GPU, y en la Sección A.4.2 para resolver consultas *k*NN.

A.4.1. Consultas por Rango en GPU

A continuación se proponen y describen el mapeo e implementación de tres algoritmos para resolver consultas por rango en GPU: un algoritmo de fuerza bruta y dos algoritmos de búsqueda basados en índices. Para fines de la presente tesis, en todos los algoritmos que resuelven consultas por rango, el resultado de la búsqueda es únicamente el número de elementos encontrados.

A.4.1.1. Fuerza Bruta en GPU Procesando Consultas por Rango

El propósito de esta implementación es que cada thread de un bloque resuelva la evaluación de distancia entre un elemento de la base de datos y la consulta, evitando

acceder a una estructura de datos intermedia o índice.

Previamente se almacenó la base de datos en una matriz de tamaño $D \times SIZE_{BD}$ (D=dimensión de los elementos[1], y $SIZE_{BD}$= número de elementos de la base de datos), en donde cada columna representa un elemento. El hecho de almacenar los datos por columna es con la finalidad de que threads consecutivos accedan a posiciones contiguas de memoria al leer datos desde *device memory*.

El algoritmo que implementa este método está dividido en dos etapas delimitadas por la función de sincronización `__syncthreads()`. En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory*. En la segunda etapa, los elementos de la base de datos se asignan a los threads siguiendo una distribución circular, y cada thread realiza las evaluaciones de distancia entre sus elementos y la consulta. En caso que la distancia entre el elemento y la consulta sea menor que el radio de búsqueda, el elemento es agregado al conjunto resultado.

A.4.1.2. Lista de Clusters (*LC*) en GPU Procesando Consultas por Rango

La estructura de datos usada para implementar la *LC* consistió en 3 matrices, denotadas como *CENTROS*, *RC* y *CLUSTERS*. La matriz *CENTROS* es de tamaño $D \times SIZE_{Centros}$ ($SIZE_{Centros}$=cantidad de centros de clusters), donde cada columna representa un centro de cluster. *RC* es un array de longitud $SIZE_{Centros}$ con los radios cobertores de cada cluster. *CLUSTERS* es una matriz de $D \times SIZE_{Clusters}$ ($SIZE_{Clusters}$=cantidad de elementos en todos los clusters), donde cada columna representa un elemento de cluster, con la característica que los elementos de un mismo cluster se encuentran en columnas contiguas. Al igual que el algoritmo de

[1]En el caso de la base de datos *Words D* es el tamaño máximo de una palabra del diccionario

fuerza bruta, el hecho de almacenar los datos por columnas es para favorecer la fusión de instrucciones de lecturas.

El algoritmo que implementa la búsqueda del índice *LC* en GPU está divido en etapas delimitadas por la función de sincronización `__syncthreads()`. En la primera etapa los threads de cada bloque colaboran para copiar en *shared memory* la consulta que le corresponde resolver. En la segunda etapa, los centros de clusters se asignan a los threads siguiendo una distribución circular, y cada thread realiza la evaluación de distancia entre la consulta y un subconjunto de centros. Si el centro está dentro del radio de búsqueda, éste se agrega al conjunto respuesta. En *shared memory* se almacenan los clusters sobre los que se debe realizar una búsqueda exhaustiva. Si la consulta está completamente contenida en un cluster, entonces se detiene la búsqueda, porque dada las características del índice *LC*, no habrán resultados en los siguientes clusters. Finalmente, en la tercera etapa, los elementos de los clusters son asignados a los threads siguiendo una distribución circular. Los elementos de clusters no descartados son comparados contra la consulta.

A.4.1.3. *SSS-Index* en GPU Procesando Consultas por Rango

Este índice se representó por 3 matrices denominadas *PIVOTES*, *DISTANCIAS* y *BD*. *PIVOTES* es una matriz de $D \times SIZE_{piv}$ ($SIZE_{piv}$=cantidad de pivotes), que almacena en cada columna un pivote. *DISTANCIAS* es una matriz de $SIZE_{piv} \times SIZE_{BD}$ ($SIZE_{BD}$=cantidad de elementos de la BD), que almacena las distancias entre los pivotes y los elementos de la base de datos. *BD* es una matriz de $D \times SIZE_{BD}$ que almacena en cada columna un elemento de la base de datos.

El algoritmo que implementa la búsqueda del índice *SSS-Index* en GPU está divido en etapas delimitadas la función de sincronización `__syncthreads()`. En la

primera etapa los threads del mismo bloque colaboran para copiar la consulta que le corresponde resolver a *shared memory*. En la segunda etapa, los threads del mismo bloque (siguiendo una distribución circular) obtienen la distancia entre todos los pivotes y la consulta. Esta distancia es almacenada en *shared memory*. Finalmente, en la tercera etapa, cada elemento es asignado a un thread siguiendo una distribución circular. Cada thread intenta descartar su elemento mediante desigualdad triangular, y en caso de no ser posible, el mismo thread realiza la evaluación de distancia entre el elemento y la consulta.

En el artículo [8], los autores encontraron empíricamente que la cantidad de pivotes creadas con valores alrededor de $\alpha = 0{,}4$ produce el óptimo para este índice. Sin embargo, si observamos la Figura A.10, el rendimiento óptimo para el *SSS-Index* sobre GPU se consigue con $\alpha = 0{,}6$ (1 pivote) para la base de datos de vectores. Esta figura muestra tres gráficos correspondientes al tiempo de ejecución, la cantidad de lecturas de 32, 64 y 128 bytes en *device memory* y el promedio de evaluaciones de distancia por consulta, para el *SSS-Index* sobre la base de datos *Images* utilizando distintos valores de α. Los valores fueron normalizados al mayor valor observado en el experimento.

Como se esperaba, mientras mayor es el α, mayor es la cantidad de evaluaciones de distancia realizadas. Pero el mejor rendimiento en cuanto a tiempo de ejecución se consigue con $\alpha = 0{,}66$ (1 pivote), a pesar de realizar 17.7 veces más evaluaciones de distancia que usando $\alpha = 0{,}5$ (73 pivotes). La respuesta a este comportamiento lo tiene el gráfico de operaciones de lecturas/escrituras. Cuando se utilizan más pivotes, los threads de un *warp* son más propensos a divergir. Por consiguiente, el patrón de acceso a memoria es más irregular impidiendo la fusión de operaciones de lectura/escritura.

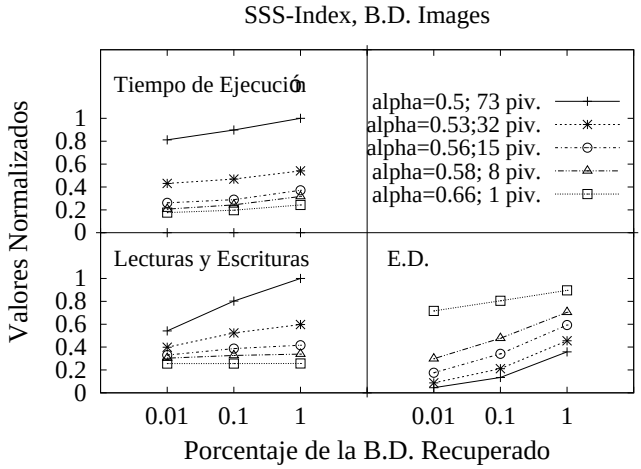


Figura A.10: Valores normalizados del tiempos de ejecución, cantidad de lecturas/escrituras (de 32, 64 o 128 bytes) a *device memory* y del promedio de evaluaciones de distancia por consulta del *SSS-Index* sobre GPU para la base de datos *Images*.

Esto último significa que en este caso, realizar menos evaluaciones de distancia no compensa el costo causado por la divergencia en la secuencia de instrucciones de threads del mismo *warp* y la irregularidad en los accesos a memoria.

A.4.1.4. Resultados Experimentales sobre GPU para Resolver Consultas por Rango

Los experimentos fueron realizados sobre un servidor que posee una GPU NVIDIA Tesla T2070, con 14 multiprocesadores, 32 núcleos por multiprocesador y 48K de *shared memory*. El tamaño de *device memory* es de 5GB. Este servidor posee 2 CPUs Intel's Nehalem Xeon E5645 con 24GB de RAM, y con un total de 12 núcleos, donde cada uno de ellos implementa tecnología Intel Hyperthreading. Los experimentos sobre OpenMP y versiones secuenciales fueron ejecutados sobre este mismo servidor.

Se usaron las mismas bases de datos mostradas en la Sección A.3: *Words* e *Images*. Pero, en esta sección se agregó una base de datos más, con elementos de alta dimensión, llamada *Faces*.

***Faces*:** Esta base de datos es una colección de 8,480 rostros, obtenidos de Face Recognition Grand Challenge [50]. Se aplicó el método Eigen Face [54] para obtener una matriz de proyección, la que puede ser usada para generar un vector característica a partir de cualquier imagen de rostro. Se usó esta colección como una distribución de probabilidades empírica, de la cual generamos una colección de elementos de imágenes de rostros aleatorios, conteniendo 95,325 objetos de dimensión 254. Desde la misma base de datos se tomaron 23,831 elementos que se utilizaron como consultas. Se usó la distancia euclidiana como función de distancia. Se usaron como radios, aquellos que permiten recuperar el 0.01 %, 0.1 % y 1 % de elementos de la base de datos por consulta.

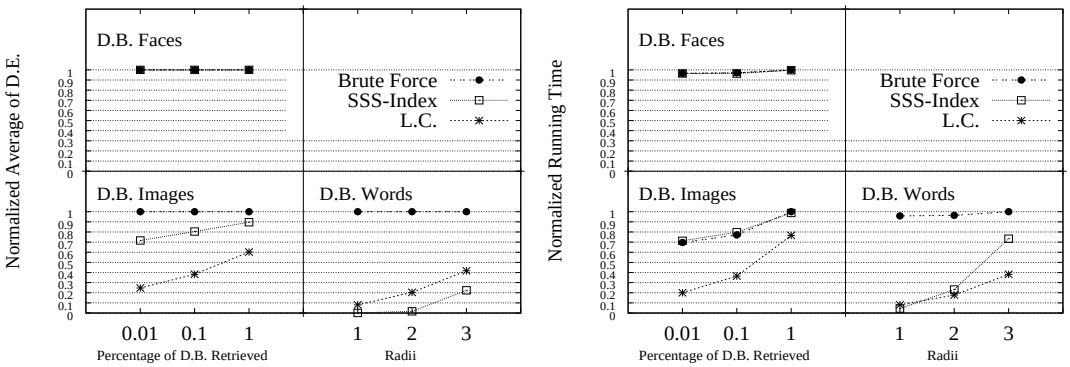
La Figura A.11 muestra tres gráficos correspondientes al tiempo de ejecución, cantidad de lecturas/escrituras a *device memory* y el promedio de evaluaciones de distancia por consulta de los algoritmos *Fuerza Bruta*, *LC* y *SSS-Index*. Todos los valores fueron normalizados al mayor valor observado en el experimento. Se observa que el comportamiento no es el mismo para todas las bases de datos. Para la base de datos *Faces*, los resultados son siempre iguales. Esto se debe a que todos los métodos acceden al mismo número de elementos: a todos ellos. La base de datos *Faces*, está compuesta por elementos de alta dimensión, y como es sabido ([17]) los índices son ineficientes para descartar elementos en este tipo de espacios, por lo que terminan accediendo a todos los elementos de la base de datos, al igual que el algoritmo de fuerza bruta.

Observando los resultados de la Figura A.11(a), como se esperaba, los índices

presentan un reducido número de evaluaciones de distancia comparado con el algoritmo de fuerza bruta para las bases de datos *Images* y *Words*. En computación secuencial, debido al alto costo de una evaluaciones de distancia, el número de estas últimas determina el comportamiento del tiempo de ejecución ([36, 24]), por consiguiente se esperaría que el tiempo de ejecución mostrado por la Figura A.11(b) imitara el comportamiento de la Figura A.11(a) correspondiente a las evaluaciones de distancia. Pero, los resultados de tiempo de ejecución contradicen parcialmente esta intuición. La Figura A.11(c) tiene la respuesta: el patrón de acceso a memoria, el que influye en gran medida al rendimiento de las actuales GPUs, se comporta mejor para el índice *LC*. La regularidad del algoritmo de búsqueda del índice *LC* conduce a un mejor patrón de acceso a los datos en la memoria de la GPU.

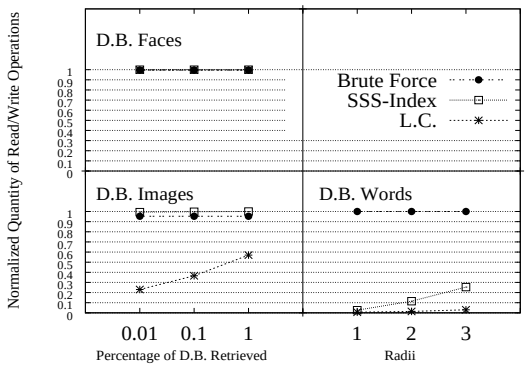
Pero, a pesar de lo indicado anteriormente, hay una excepción en el tiempo de ejecución de la base de datos *Words* con radio 1, en donde el *SSS-Index* realiza más operaciones de lectura y escritura, y sin embargo obtiene un mejor tiempo de ejecución. Esto se explica porque en este caso particular la cantidad de operaciones de lectura/escritura fue suficientemente pequeño para que otros factores tuvieran influencia. Factores tales como, la cantidad de registros utilizados por los threads en el *SSS-Index* fue un 12 % menor, y debido a aquello el número de warps activos por multiprocesador es mayor. También, el número total de instrucciones ejecutadas por el *SSS-Index* fue 2 % menor, y el número de warps que tuvieron que serializar su acceso a shared memory (debido a conflictos de acceso al mismo banco) fue 34 % menor en el *SSS-Index*. Pero, todos estos factores se vuelven irrelevantes cuando el número de operaciones de lectura/escritura se incrementa, debido al alto costo asociado a estas últimas operaciones.

La coalescencia y el alineamiento en los accesos a memoria influyen en gran



(a) Evaluaciones de Distancia

(b) Tiempo de Ejecución



(c) Operaciones de Lectura/Escritura

Figura A.11: Valores normalizados del **a)** promedio de evaluaciones de distancia por consulta, **b)** tiempo de ejecución, y **c)** operaciones de lectura/escritura. Se usó una GPU, procesando consultas por rango.

medida al rendimiento de las actuales GPUs. Cuando un *warp* ejecuta accesos a memoria no alineados o no consecutivos, el hardware no es capaz de fusionarlos y una referencia a memoria podría convertirse en varios accesos separados. Cuando se activa el *CUDA profiler*, se puede observar que el número de operaciones de lectura/escritura del *SSS-Index* crece mucho más que en el caso del *LC*, debido a las operaciones de memoria no coalescentes. Esto último y lo regular del código del algoritmo de búsqueda del *LC*, explican su rendimiento superior sobre las demás

implementaciones.

Con respecto a los resultados sobre la base de datos *Faces*, se observó un comportamiento diferente. Este es un espacio de alta dimensión, suficientemente alto para que los índices no sean capaces de descartar elementos, siendo el algoritmo de *Fuerza Bruta* el método más eficiente. Los índices intentan descartar elementos, pero no pueden, por lo tanto terminan comparando todos los elementos de la base de datos contra la consulta, que es exactamente lo que realiza el algoritmo de *Fuerza Bruta*, pero los índices ejecutan muchas más instrucciones al aplicar el método de búsqueda y descarte del índice. Este problema es conocido como la *maldición de la dimensionalidad* ([5]): en espacios de alta dimensión, los métodos de indexación en espacios métricos pierden eficiencia.

Enfocándonos en el rendimiento general, la Figura A.12 muestra el speed-up de todas nuestras implementaciones, tomando como referencia el algoritmo de fuerza bruta secuencial (ejecutado sobre un único procesador). En este experimento se utilizó la base de datos *Words* de 100,000 elementos, obteniendo resultados similares para la base de datos *Images*.

Las primeras dos columnas de la Figura A.12 muestran el speed-up de las implementaciones secuenciales utilizando los índices. Las siguientes dos columnas muestran el speed-up de las versiones multi-core de los índices sobre el servidor Xeon. Finalmente, las últimas dos columnas muestran el speed-up de nuestras implementaciones en GPU. Dada la gran variación en los resultados para distintos radios, se muestran los resultados en dos gráficos.

En general, el índice *LC* en GPU supera ampliamente al resto, lo que era de esperarse debido a su buena regularidad en el algoritmo de búsqueda y su patrón de acceso a *device memory*. Un comportamiento común y esperado es, mientras más

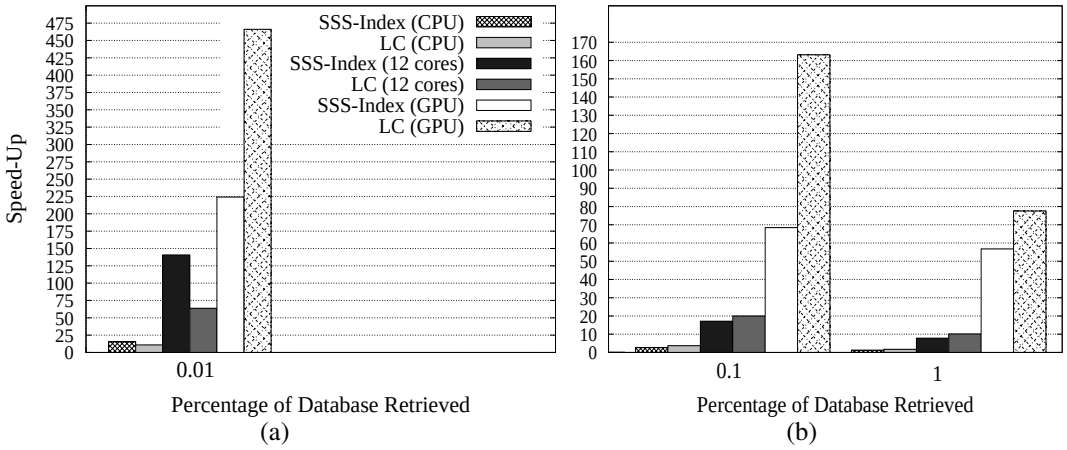


Figura A.12: Speed-up de los índices *SSS-Index* y *LC* usando diferentes plataformas (con la base de datos *Words* de 100,000 elementos). Los valores fueron calculados sobre el algoritmo de fuerza bruta secuencial, procesando consultas por rango.

pequeño es el radio, mayor es el beneficio de los índices (secuencial o paralelo). Recordemos que el caso de referencia para calcular el speed-up es el algoritmo de fuerza bruta secuencial, es decir, todas las posibles evaluaciones de distancia son ejecutadas. Si se incrementa el radio de búsqueda, el número de evaluaciones de distancia se incrementa por parte de los índices, acortando la brecha con el algoritmo de fuerza bruta. A pesar que el *SSS-Index* no muestra el mejor rendimiento, es menos afectado que el *LC* cuando el radio de búsqueda crece. Esto último es principalmente porque la irregularidad en el algoritmo de búsqueda del *SSS-Index* es poco afectada al incrementar el radio.

Para algunos lectores, los speed-up de las versiones en GPU podrían parecer no muy impresionantes, dado que nuestra GPU cuenta con 14 multiprocesadores (y 448 núcleos), comparados con el servidor utilizado para las versiones multi-core de 12 núcleos Xeon. Sin embargo, es importante recordar que cada núcleo de NVIDIA

es extremadamente más simple que un núcleo Intel basado en microarquitectura Nehalem; el paralelismo a nivel de instrucción es casi no es explotado, al contrario que en los Intel CPU-cores donde representa la principal razón de rendimiento para procesadores más complejos.

A.4.2. Consultas *k*NN en GPU

A pesar de utilizar como base la solución de consultas por rango para resolver consultas *k*NN, se encontraron diferentes dificultades a la hora de implementar y mapear los índices a GPU para resolver consultas *k*NN. Esto último es debido principalmente a las particulares características de la GPU y su jerarquía de memoria.

A continuación se describen nuestras propuestas de indexación y fuerza bruta para resolver consultas *k*NN en GPU. Tal como se realizó en la solución de consultas por rango, aquí también se resuelve un conjunto de consultas en paralelo, cada una de ellas en un diferente CUDA Block, y cada consulta con un conjunto de threads.

A.4.2.1. Búsqueda Exhaustiva

En esta sección se proponen dos algoritmos de búsqueda exhaustiva para resolver consultas de tipo *k*NN. El primero está basado en trabajos previos ([10, 26, 30]), donde se calculan todas las distancias entre los elementos de la base de datos y la consulta, y luego estas distancias se ordenan para seleccionar los primeros K como el resultado final. En el segundo algoritmo se propone que cada consulta sea resuelta por un diferente CUDA Block, y que cada uno de ellos utilize un conjunto de heaps [29] para mantener los K elementos más cercanos a la consulta a través del proceso de búsqueda.

Ambos algoritmos reciben como parámetro de entrada el array de distancias δ, en donde δ[i] es la distancia entre el i-ésimo elemento de la base de datos y la consulta. Para obtener el array δ se debe lanzar un kernel previamente, en el que cada thread calcula las distancias entre un grupo de elementos de la base de datos y la consulta. A continuación se describen los métodos basado en ordenamiento y basado en heaps respectivamente.

Procesamiento Basado en Ordenamiento

En este método se propone ordenar los elementos del array δ, el que es recibido como parámetro de entrada y contiene las distancias de todos los elementos de la base de datos contra la consulta. El resultado final son los *K* primeros elementos del array ordenado. Para implementar esta estrategia en GPU, se necesita un algoritmo paralelo eficiente de ordenamiento. Cederman y Philips proponen en [11] una implementación de quicksort llamada *GPU-Quicksort*, la que muestra resultados más eficientes que previos algoritmos de ordenamiento en GPU como *Radix-sort* [53] o *insertion sort* [26]. Por esto último, se utilizó el *GPU-Quicksort* como algoritmo de ordenamiento del array δ.

Procesamiento Basado en Heaps

En esta sección se propone que cada consulta sea resuelta por un CUDA Block distinto, en donde cada thread del CUDA Block utiliza un heap [29] en *device memory* para mantener los *K* elementos más cercanos a la consulta a través del proceso de búsqueda de la consulta *k*NN.

El algoritmo de búsqueda de este método está dividido en dos etapas. En la primera etapa, todos los threads cooperan para copiar la consulta a *shared memory*

(pequeña memoria de baja latencia ubicada dentro del multiprocesador). En la segunda etapa los threads (siguiendo una distribución circular) visitan los elementos del array δ e intentan insertar el elemento en su heap correspondiente. Para ello, es necesario asignar a cada thread (de todos los CUDA Blocks) un heap de tamaño K en *device memory*. Este conjunto de threads es almacenado en una matriz de tamaño $K \times T$, donde T es el número total de threads (tomando en cuenta todos los CUDA Blocks). Cada columna de esta matriz representa un heap. El hecho de almacenar los heaps por columnas es para favorecer la coalescencia de operaciones de lectura/escritura.

Después que cada CUDA Block ha visitado todos los elementos del array δ, cada thread del bloque tiene sus K elementos más cercanos a la consulta almacenados en su heap en *device memory*. Luego, se aplica un proceso de reducción, que se compone de dos etapas ilustradas en la Figura A.13, donde cada triángulo representa un heap. En la primera etapa, los threads del primer warp del CUDA Block acceden (circularmente) a los elementos de los heaps anteriores, pero esta vez los elementos son guardados en heaps almacenados en *shared memory*. En la segunda etapa, el primer thread del warp accede a los elementos de los heaps de la etapa previa, y almacena los K elementos finales en un heap almacenado también en *shared memory*.

Cabe destacar que los heaps del mismo warp tienen sus elementos raíces en direcciones de memoria consecutiva. De este modo, estamos favoreciendo la coalescencia de operaciones de lectura y escritura.

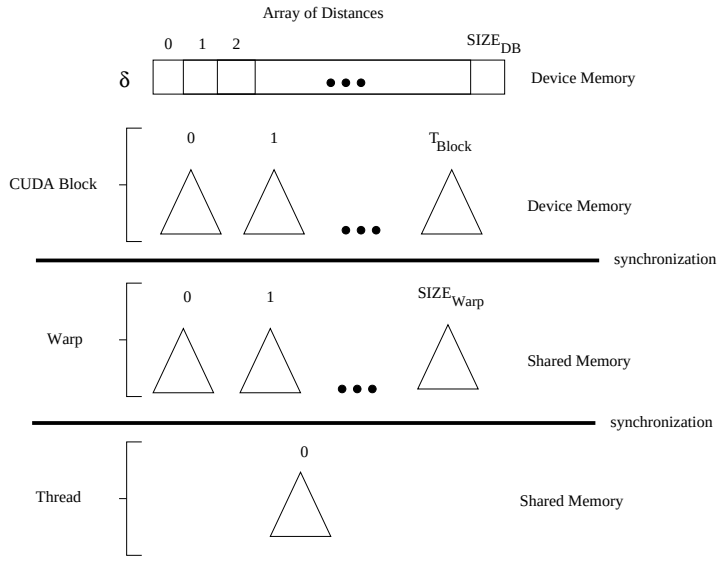


Figura A.13: Ilustración de los pasos para reducir el array de distancias δ a los K resultados finales. Cada triángulo representa un heap.

A.4.2.2. Lista de Clusters (LC) en GPU procesando consultas *k*NN

En esta sección se utilizaron las mismas estructuras usadas para resolver consultas por rango (Sección A.4.1.2), es decir, las matrices *CENTERS*, *RC* y *CLUSTERS*, que almacenan los centros, radios cobertores y elementos de clusters respectivamente. Dichos elementos son almacenados por columnas en las matrices para favorecer al acceso contiguo de memoria y mejorar la coalescencia de operaciones de lectura y escritura.

Como se describió en la Sección A.2.1, hay 2 métodos para procesar consultas *k*NN: el método de rango decreciente, y de rango creciente. El método de rango decreciente ha mostrado mejores resultados en computación secuencial ([52, 18]), pero en nuestra exploración hemos encontrado que el método de rango creciente es más adecuado para el uso de GPUs. La Figura A.14 muestra el tiempo de ejecución, evaluaciones de distancia y cantidad de operaciones de lectura y escritura del *LC* en

GPU, usando ambos métodos sobre la base de datos *Images*.

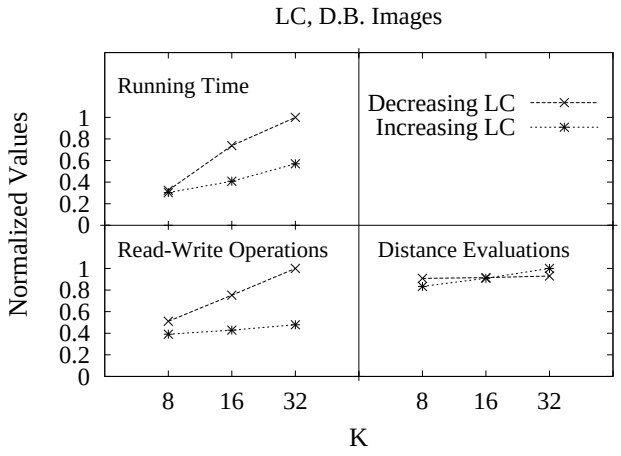


Figura A.14: Valores normalizados del tiempo de ejecución, evaluaciones de distancia y cantidad de operaciones de lectura y escritura, de los métodos de rango decreciente y creciente, procesando consultas *k*NN sobre la base de datos *Images*, con el índice *LC*.

La principal razón del bajo rendimiento del método de rango decreciente en GPU, es que todos los threads de un CUDA Block cooperan en la solución de una consulta, y este paralelismo intra-consulta no permite reducir el radio suficientemente rápido. En computación secuencial, el método de rango decreciente ajusta el radio tras visitar cada elemento de la base de datos, pero en GPU se tiene un conjunto de cientos de threads procesando una consulta en paralelo, por lo que mantener un radio global de búsqueda implica un alto número de sincronizaciones. Si uno quiere evitar sincronizaciones, de forma que cada thread use un radio local, la calidad de éste es baja, y no permite un buen descarte, aumentando en gran medida el número de evaluaciones de distancia y el tiempo de ejecución. Por otro lado, el método de rango creciente se adapta bien a las características de la GPU por varias razones: (1) el hecho que cada búsqueda por rango sea ejecutada con

el mismo rango para todos los threads, mejora la regularidad en el código; (2) la regularidad en el código incrementa la coalescencia de operaciones de lectura y escritura; (3) este método implica menor número de sincronizaciones que el método de rango decreciente.

También, en el índice *LC*, cada thread utiliza un heap para almacenar los K elementos resultados a través de la búsqueda. Si no es posible el descarte de un elemento con el método de búsqueda del *LC*, se intenta insertar dicho elemento al heap del thread, lo que ocurre sólo si la distancia del elemento a la consulta es menor que la que posee la raíz del heap. Por lo tanto, tras procesar la consulta con el *LC*, cada thread del CUDA Block tendrá un heap (en *device memory*) con sus K elementos más cercanos a la consulta. Luego, estos elementos son reducidos a un sólo heap en *shared memory* usando los mismos pasos del algoritmo exhaustivo basado en heaps (Sección 4.2.1.2).

A.4.2.3. SSS-Index on a single GPU processing *k*NN queries

En esta sección se utilizaron las mismas estructuras usadas para resolver consultas por rango (Sección A.4.1.3), es decir, las matrices *PIVOTS*, *DISTANCES* y *DB*, que almacenan los pivotes, distancias entre pivotes y elementos y los mismos elementos de la base de datos respectivamente. Al igual que en secciones previas, los elementos son almacenados por columnas.

Procesando consultas *k*NN, se encontró el mismo comportamiento al aumentar el parámetro α que el observado en la Sección A.4.1.3, donde con 1 pivote se logra el mejor rendimiento en GPU sobre bases de datos de vectores.

En este índice, al igual que en los métodos anteriores, cada thread utiliza un heap como estructura auxiliar. Se aplica el método de búsqueda del *SSS-Index*, y

posteriormente se intentan insertar los elementos no descartados en los heaps, para posteriormente reducir dichos heaps a sólo uno, almacenado en *shared memory*, con el resultado final.

A.4.2.4. Resultados Experimentales sobre GPU para Resolver Consultas *k*NN

Los experimentos fueron realizados sobre la misma tarjeta gráfica y CPU que los usados para resolver consultas por rango (Sección A.4.1.4). Los experimentos sobre OpenMP y versiones secuenciales fueron ejecutados sobre este mismo servidor. También, las bases de datos utilizadas fueron las mismas que se utilizaron para resolver consultas rango: *Words*, *Images* y *Faces*.

Experimentos de los Métodos de Búsqueda Exhaustiva

La Figura A.15 compara los diferentes métodos de búsqueda exhaustiva descritos en la Sección A.4.2.1. *Sort-based Reduction* representa el método basado en ordenamiento. Este último método utiliza completamente la GPU para ordenar el array de distancias, por lo tanto, las consultas son resueltas una a la vez. *Batch-Heap Reduction* corresponde a nuestra propuesta de búsqueda exhaustiva basada en heaps, donde cada consulta se resuelve con un CUDA Block distinto, y múltiples consultas son resueltas en paralelo. Finalmente, se incluyó un tercer método llamado *Heap-Reduction*, que es similar al método anterior basado en heaps, pero resuelve sólo una consulta a la vez, usando sólo un CUDA Block. Este último método fue agregado para observar cómo el método basado en heaps escala desde usar sólo un CUDA Block a usarlos todos. La Figura A.15(a) compara el tiempo de ejecución para diferentes valores de K. Los valores están normalizados al mayor valor observado en el experimento. La Figura A.15(b) muestra el tiempo de ejecución acumulado de los

mismos métodos. Los experimentos fueron ejecutados con la base de datos *Faces*, dado que los algoritmos de búsqueda exhaustiva muestran resultados competitivos en espacios de alta dimensión ([17]).

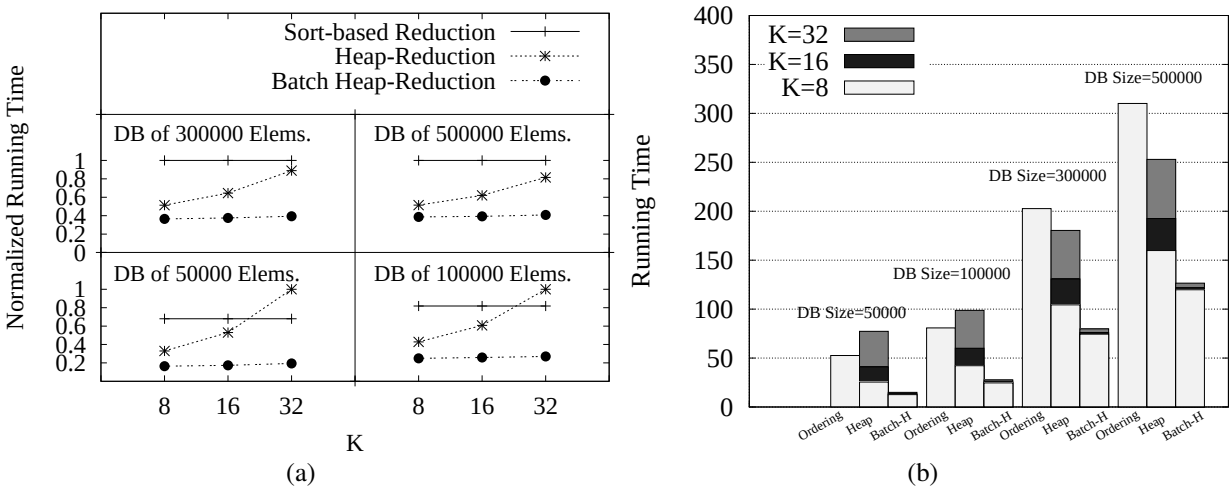


Figura A.15: **a)** Tiempo de ejecución normalizado, y **b)** Tiempo de ejecución acumulado para los diferentes algoritmos de búsqueda exhaustiva, usando diferentes K con diferentes tamaños de la base de datos *Faces*.

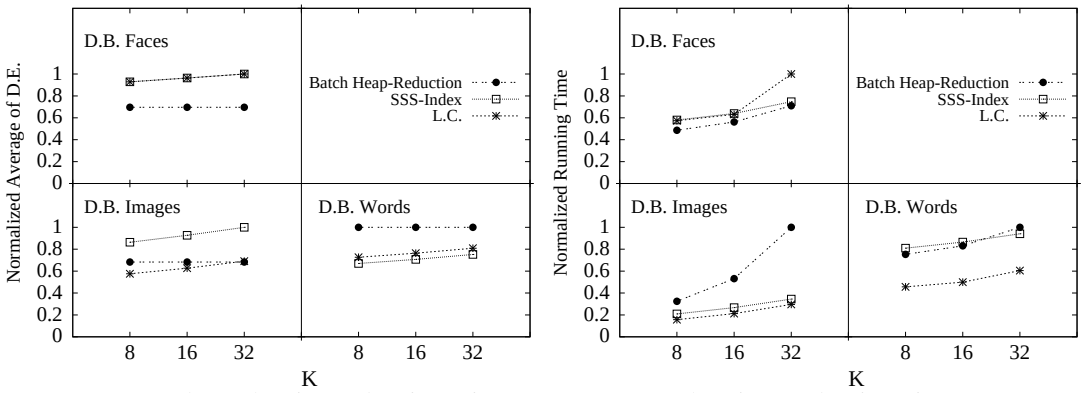
Nuestras propuestas superan en rendimiento al método basado en ordenamiento en casi todos los casos. Aún con nuestro método *Heap-Reduction*, que resuelve sólo una consulta a la vez con un CUDA Block, somos capaces de obtener mejores resultados para bases de datos de gran tamaño y pequeños valores de K, debido al mejor uso de los accesos a memoria. Cuando usamos la GPU lanzando tantos CUDA Blocks como consultas, la diferencia se incrementa, y aún más importante, nuestro método pasa a ser menos sensible al valor de K. Cabe destacar que el rendimiento de cualquier método basado en ordenamiento es independiente del valor de K, pues la diferencia radica sólo en elegir los K primeros elementos del array de distancias ordenado.

Experimentos para Búsqueda Indexada

A continuación presentamos los experimentos para procesar consultas *k*NN usando los índices *LC* y *SSS-Index*, descritos en las Secciones A.4.2.2 y A.4.2.3 respectivamente. Empíricamente se encontró que un tamaño conveniente de elementos por cluster es 64 para la bases de datos de vectores y 32 para la base de datos de palabras. Se usó el método de rango creciente, por las razones descritas en A.4.2.2. Por las razones descritas en la Sección A.4.2.3, el *SSS-Index* implementa sólo 1 pivote para las bases de datos de vectores y 64 para palabras.

La Figura A.16 muestra diferentes resultados para ilustrar varios puntos importantes de nuestros tres métodos. Los valores están normalizados al mayor valor observado del experimento. Observando la Figura A.16(a), la base de datos *Words* se comporta como se esperaba: los índices reducen el número de evaluaciones de distancia comparado con un método de búsqueda exhaustiva. Sin embargo, esto último no ocurre con las bases de datos de vectores. Este hecho implica que algunas evaluaciones de distancia de ejecutaron más de una vez por parte de los índices, lo que es posible ya que se utiliza el método de rango creciente. Cabe destacar que intencionalmente se decidió no reusar distancias calculadas previamente porque aquello agrega una gran cantidad de irregularidad en el código decrementando el rendimiento.

Era de esperar que los resultados mostrados por el gráfico de evaluaciones de distancia se replicaran en el tiempo de ejecución, pero la Figura A.16(b) contradice parcialmente esta intuición. La Figura A.16(c) tiene la explicación: el patrón de acceso a memoria, que influye en gran medida en el rendimiento de las actuales GPUs, se comporta mejor para los índices, especialmente para el *LC*. En todas nuestras



(a) Número de Evaluaciones de Distancia

(b) Tiempo de Ejecución

(c) Número de Operaciones de Lectura/Escritura

Figura A.16: Valores normalizados de **a)** evaluaciones de distancia, **b)** tiempo de ejcución, y **c)** operaciones de lectura/escritura a *device memory*.

implementaciones, las inserciones en un heap implican divergencia y pérdida de localidad, lo que implica un aumento en las operaciones de lectura y escritura. Los índices ejecutan más evaluaciones de distancia pero, debido a que muchas de esas evaluaciones de distancias son calculadas varias veces, el número de inserciones en los heaps es reducido. Sin embargo, cuando la dimensión de los elementos crece (base de datos *Faces*), el alto costo de las evaluaciones de distancia comienza a contrarrestar la disminución en las inserciones en heaps.

La Figura A.17 muestra el speed-up de los índices en GPU, una versión multi-core optimizada (usando la estrategia *local* descrita en la Sección A.3), y una versión secuencial. Los valores fueron calculados sobre el algoritmo de fuerza bruta secuencial, el que usa un heap como estructura auxiliar. El índice *LC* alcanza hasta 42x de speed-up, superando considerablemente la versión multi-core.

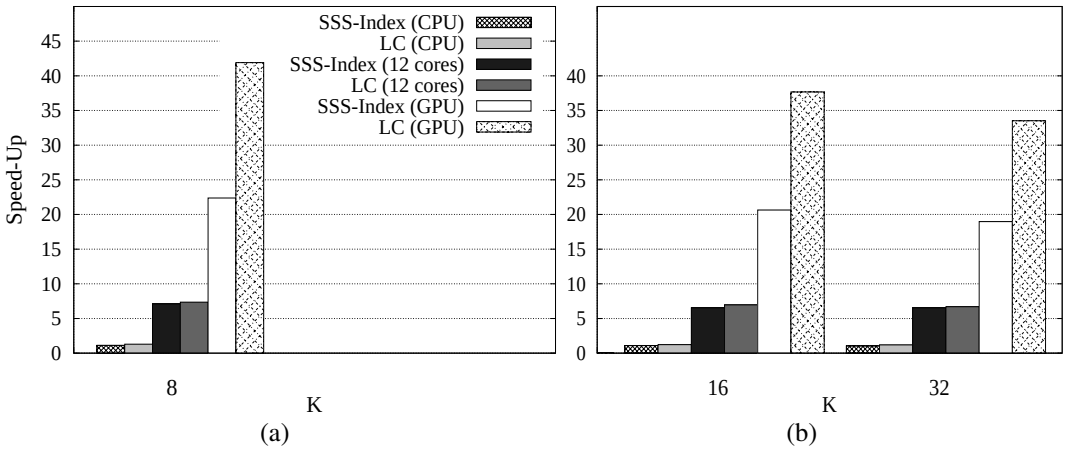


Figura A.17: Speed-up del *SSS-Index* y *LC* usando diferentes plataformas (con la base de datos *Words* de 100,000 elementos). Los valores fueron calculados sobre el algoritmo de fuerza bruta secuencial, procesando consultas *k*NN.

A.5. Estrategias de Distribución y Búsqueda sobre una Plataforma multi-GPU

Producto de los buenos resultados y altos speed-up alcanzados sobre una GPU, se modificaron los algoritmos para ser aplicados sobre una plataforma multi-GPU. Esta sección sólo se utilizó el índice *LC*, debido a sus buenos resultados sobre una GPU.

Esta sección aborda dos casos distintos de bases de datos sobre una plataforma multi-GPU. El primero, es cuando la base de datos cabe completamente en memoria de la GPU, y el segundo es cuando no cabe. A continuación se presentan ambos casos.

A.5.1. Caso 1: Base de datos en Memoria

A continuación se presentan dos estrategias llamadas *2-Stages* y *1-Stage*. La estrategia *2-Stages* está compuesta de dos etapas, en donde se establecen qué clusters deben ser procesados con qué queries y por qué procesador, y posteriormente, se comparan los clusters no descartados con las correspondientes consultas. La estrategia *1-Stage*, sólo establece un paso, donde se realiza el descarte de clusters y también la comparación entre clusters no descartados y la consulta, en el mismo kernel. Ambas estrategias asumen previamente que las consultas son resueltas por lotes.

A.5.1.1. Estrategia *2-Stages*

La idea principal es dividir el proceso de búsqueda en dos pasos: (1) intentar descartar los cluster, y los no descartados guardarlos en *device memory*; (2) cada GPU lee qué clusters debe comparar con qué queries y realiza los cálculos de distancias.

Tal como muestra la Figura A.18, la base de datos se distribuye (de manera off-line) previo al proceso de búsqueda, entre las GPUs. Se copian todos los centros de clusters ($C_1, C_2, ..., C_N$) y radios cobertores a todas las GPUs. Los elementos de los clusters, denominados como $Cluster_1, ..., Cluster_N$ en la Figura A.18 son

distribuidos circularmente entre las GPUs. Todas las GPUs tienen un mapa que les indica qué cluster está almacenado en qué GPU. Por lo anterior, cada GPU puede averiguar cuáles son clusters que deben ser comparados con una consulta dada, por lo que es necesario enviar una consulta a sólo una GPU.

En la primera etapa, llamada *Setting Scheduling* en la Figura A.18, las consultas se distribuyen entre las GPUs. Todos los threads de un CUDA Block cooperan para resolver una consulta. Cada GPU almacena en una matriz, para cada consulta, qué clusters no están descartados y qué GPU debe realizar la comparación. Dicha matriz debe ser almacenada en *device memory*, pues es un parámetro de entrada para la siguiente etapa. En la segunda etapa, llamada *Applying Scheduling* en la Figura A.18, cada GPU lee desde *device memory* una lista de tuplas, donde cada tupla está formada como $<$*consulta, lista de clusters no descartados*$>$. Cada tupla indica qué consulta debe ser procesada con qué clusters.

A.5.1.2. Estrategia *1-Stage*

Esta estrategia propone resolver las consultas en sólo una etapa, y evitar escribir en *device memory* datos para sincronizar distintas etapas como en *2-Stages*. Como se observa en la Figura A.18, los clusters son completamente distribuidos entre las GPUs, es decir, los centros $C_1, C_2, ..., C_N$, sus respectivos radios cobertores y elementos de clusters ($Cluster_1, ..., Cluster_N$) son distribuidos entre las GPUs. Por ende, cada consulta debe ser procesada por todas las GPUs.

Una vez que una consulta es enviada a una GPU, ésta aplica el algoritmo para una GPU descrito en la Sección A.4.1.2, con lo que se resuelve la consulta con sólo un kernel. Cada CUDA Block se encarga de una consulta distinta, y el kernel es lanzado con tantos CUDA Blocks como consultas.

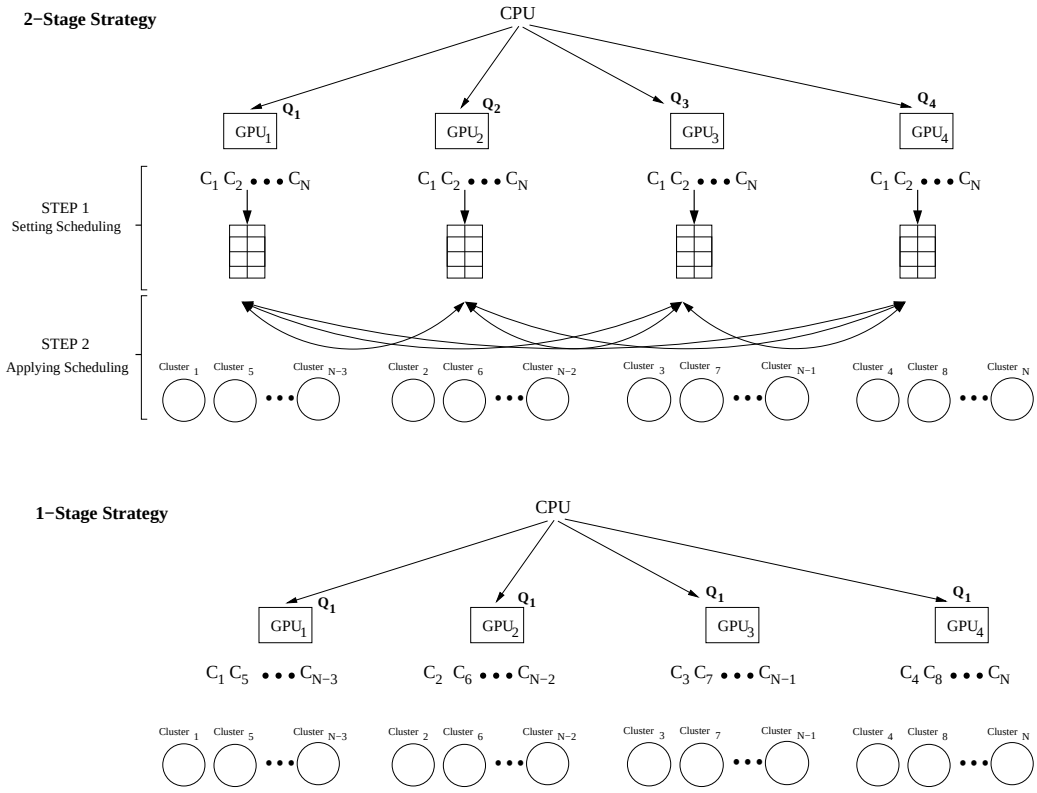


Figura A.18: Ilustración de las estrategias multi-GPU *2-Stages* y *1-Stage*.

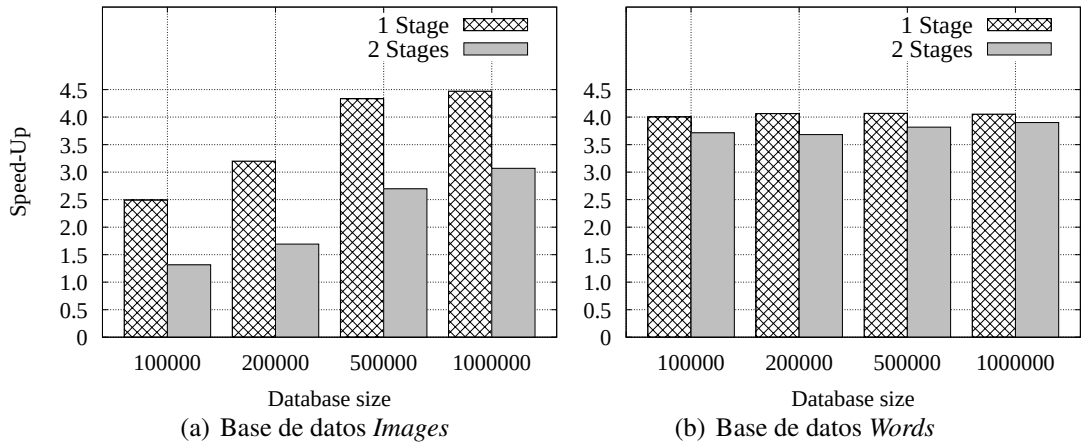
Una ventaja de esta estrategia es que reduce el número de kernels y accesos a *device memory*, porque los clusters que deben ser accedidos son conocidos en el mismo kernel. Pero, una desventaja es que siempre una consulta debe ser procesada por todas las GPUs, y además no es tan eficiente como *2-Stage* para detener una búsqueda cuando un consulta está completamente contenida en un cluster.

A.5.1.3. Resultados Experimentales: Base de Datos Cabe en Memoria

Se utilizó un servidor con 4 GPUs. Cada GPU es una NVIDIA Tesla C1060 con 30 multiprocesadores, 8 núcleos por multiprocesador, 16KB de memoria compartida, y 4GB de *device memory*. Se usaron versiones extendidas de las bases de datos

Words e *Images*.

La estrategia *2-Stages* está diseñada para mejorar el balance de carga entre las GPUs, pero aún cumpliendo aquello, introduce comunicación CPU-GPU extra. Por otro lado, *1-Stage* no implica inter-comunicación entre GPUs, y la única penalización en comunicación relevante es la potencial copia innecesaria a ciertas GPUs. Esto último explica el mejor rendimiento de la estrategia *1-Stage* sobre *2-Stages* mostrado en la Figura A.19. Esta figura muestra el speed-up de ambas estrategias multi-GPU sobre la versión de 1 GPU del *LC*, donde la versión de 1 GPU está ejecutada en la misma tarjeta gráfica que las versiones multi-GPU.



(a) Base de datos *Images* (b) Base de datos *Words*

Figura A.19: Speed-up de las estrategias multi-GPU sobre la versión de 1 GPU del *LC*. Todas las versiones se ejecutaron sobre el mismo modelo de tarjeta gráfica (Tesla C1060).

La estrategia *1-Stage* escala muy bien con el tamaño de la base de datos, e incluso se alcanza un speed-up superlineal (4.5x de speed-up con 4 GPUs) para la base de datos *Images*. Este comportamiento se explica por el *occupancy*, que es un indicador de la cantidad de threads activos por multiprocesador. La cantidad

de memoria compartida en el algoritmo de búsqueda del *LC* es proporcional a la cantidad de centros, y la estrategia *1-Stage* divide los centros entre las GPUs, por lo que disminuye la memoria compartida usada, y esto incrementa el *occupancy*. En cambio, en *2-Stages* todas las GPUs tienen todos los centros.

Un importante indicador a tomar en cuenta, especialmente en motores de búsqueda Web es el *throughput*, que es el número de consultas resueltas por unidad de tiempo. La Figura A.20 muestra el máximo throughput de *1-Stage* sobre nuestra plataforma muti-core de 4 GPUs. Se alcanza hasta 80,794 consultas resueltas por segundo. Podría parecer poco para condiciones de alto tráfico de consultas, pero es alcanzado con sólo 4 GPUs, y la estrategia *1-Stage* escala muy bien cuando se incrementa el número de GPUs y también el tamaño de la base de datos.

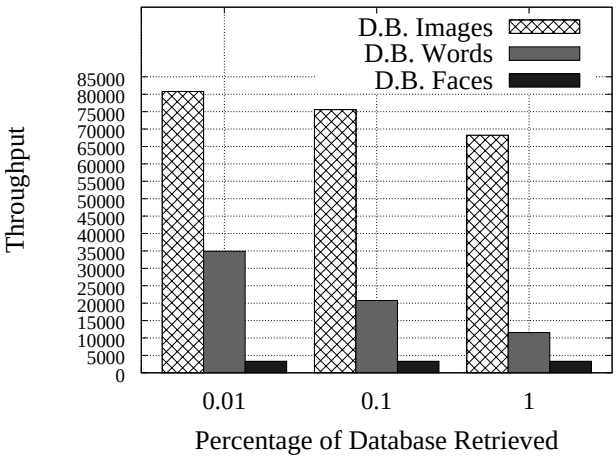


Figura A.20: Throughput para la estrategia *1-Stage*, sobre la plataforma multi-core de 4 GPUs.

A.5.2. Caso 2: Base de Datos No Cabe en Memoria

En esta sección se proponen y comparan diferentes algoritmos y estrategias para resolver consultas por similitud, sobre bases de datos suficientemente grandes como para no caber en *device memory*. Este es el caso de la mayoría de las bases de datos reales en producción, donde la memoria de la GPU (una o varias) no es suficiente para almacenar la base de datos completamente. En primer lugar, se propone un nuevo índice jerárquico por niveles, llamado *Lista de Superclusters* (LSC), que es una variante del *LC*. Posteriormente, se proponen dos pipelines: (1) un pipeline híbrido entre la CPU y GPU, y (2) un pipeline entre transferencias CPU-GPU y kernels.

A.5.2.1. Lista de Superclusters (*LSC*) en GPU

En esta sección se propone una variante del índice *LC*, denominado *Lista de Superclusters*, que es un índice jerárquico multi-nivel (que toma en cuenta la organización de la memoria en GPU). Su construcción tiene dos etapas. En la primera, se obtienen S clusters de tamaño K, usando el algoritmo de construcción del *LC*. Estos primeros clusters componen el primer nivel de la jerarquía, y son denominados superclusters. En la segunda etapa, se crea un índice *LC* dentro de cada supercluster, con sus propios elementos.

En este índice la unidad mínima de transferencia CPU-GPU es un supercluster. Tras cargar un supercluster en *device memory*, un kernel es lanzado para procesarlo con Q CUDA Blocks, donde Q es la cantidad de consultas a procesar. El kernel de búsqueda posee tres etapas: (1) Los primeros S threads calculan las distancias entre los superclusters y las consultas; (2) con las distancias del paso anterior se

intenta descartar cada supercluster, y por ende todos sus elementos; (3) por cada supercluster no descartado, se aplica el algoritmo de búsqueda en el *LC* dentro de él, siguiendo la estrategia *1-Stage* (Sección A.5.1.2).

A.5.2.2. Pipeline CPU-GPU

Con el objetivo de minimizar el número de transferencias a GPU y de incrementar el grado de paralelismo, se propone en esta sección un pipeline híbrido entre CPU y GPU. La CPU intenta descartar clusters (o superclusters en el caso del *LSC*), mientras la GPU procesa clusters no descartados. Este pipeline se implementó para ambos índices *LC* y *LSC*.

Si N es la cantidad de clusters que caben en *device memory*, y Q es la cantidad de consultas, los pasos del pipeline (mostrados por la Figura A.21) son los siguientes: (1) se distribuyen los clusters (o supercluster en el caso del *LSC*) de acuerdo a la estrategia *1-Stage* (Sección A.5.1.2), y se intentan descartar N clusters (o un supercluster de N clusters) con threads en CPU, donde un cluster es descartado sólo si no intersecta con ninguna de las Q consultas; (2) se copian los ID de los clusters no descartados; (3) se copian los clusters a *device memory* y se lanza un kernel para procesarlos con las Q consultas correspondientes. Tomando en cuenta que sólo se necesita un thread de CPU para manejar una GPU (paso 3 en la Figura A.21), mientras el tercer paso está en ejecución, el primer paso (con los demás threads de CPU) está en ejecución también, pero intentando descartar los siguientes N clusters (o superclusters).

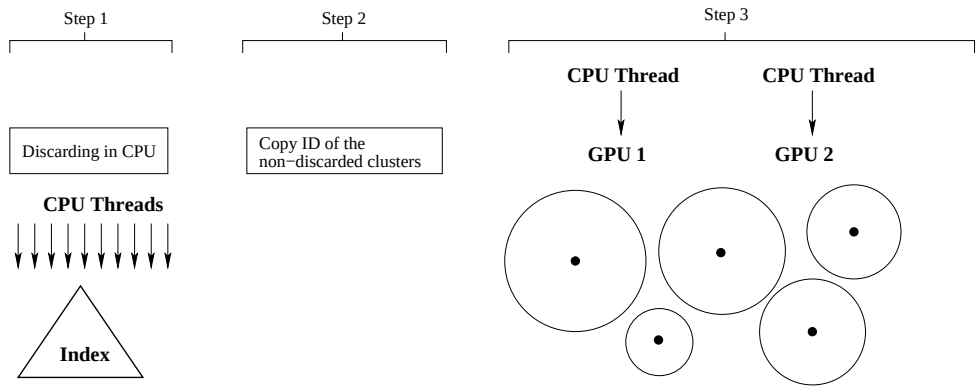


Figura A.21: Esquema del pipeline CPU-GPU.

A.5.2.3. Pipeline entre Copias Asíncronas y Kernels

La función `cudaMemcpyAsync` permite ejecutar transferencias hacia (y desde) *device memory* mientras un kernel está en ejecución. Esto es posible mediante el uso de *CUDA streams*, donde cada CUDA stream puede contener una secuencia de instrucciones distintas. Transferencias CPU-GPU y kernels de diferentes CUDA streams pueden ser ejecutados en paralelo.

Este pipeline, ilustrado por la Figura A.22 se implementó para ambos índices *LC* y *LSC*. Si N es la cantidad de clusters que caben en *device memory*, se crean dos CUDA streams, y cada uno de ellos se compone de dos pasos: (1) copiar $N/2$ clusters (o un supercluster de $N/2$ clusters) a *device memory*; (2) lanzar un kernel para procesar las consultas con los clusters correspondientes. Luego, el paso 1 copia los siguientes clusters al mismo tiempo que el paso 2 está ejecutando un kernel con los clusters previos. Sólo se crearon dos CUDA streams porque esta cantidad permite un buen balance en tiempo de ejcución entre las transferencias y las ejecuciones de kernels.

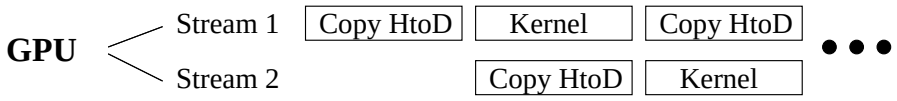


Figura A.22: Esquema del pipeline entre transferencias asíncronas y kernels.

siempre se copia un cluster del *LC* o supercluster del *LSC* con sólo una invocación a la función de copia `cudaMemcpyAsync`, porque sus elementos están contiguos en memoria; esto es clave para explotar eficientemente el gran ancho de banda entre CPU y GPU, pues pequeñas transferencias de memoria no son capaces de ocultar latencias.

A.5.2.4. Estrategia Multi-pipeline

Nuestra propuesta final combina las tres estrategias previas en una llamada estrategia *multi-pipeline*. Se utiliza el índice *LSC*, y se crean *P*threads en CPU, uno por núcleo de la CPU, dejando *G* threads a cargo de *G* GPUs ($G < P$). Mientras los núcleos de la CPU intentan descartar superclusters con el pipeline CPU-GPU (Sección A.5.2.2), cada GPU procesa superclusters no descartados usando el pipeline de copias y kernels (Sección A.5.2.3). La Figura A.23 muestra un esquema de esta estrategia.

A.5.2.5. Resultados Experimentales: Base de Datos No Cabe en Memoria

Todos los experimentos en esta sección fueron ejecutados en un servidor multicore de 2 GPUs NVIDIA Tesla M2070, donde cada una posee 14 multiprocesadores, 32 núcleos por cada multiprocesador, 48KB de memoria compartida y 5GB de *device memory*. La CPU está compuesta por dos procesadores Intel Xeon E5645 de 2.4GHz con 24GB de RAM.

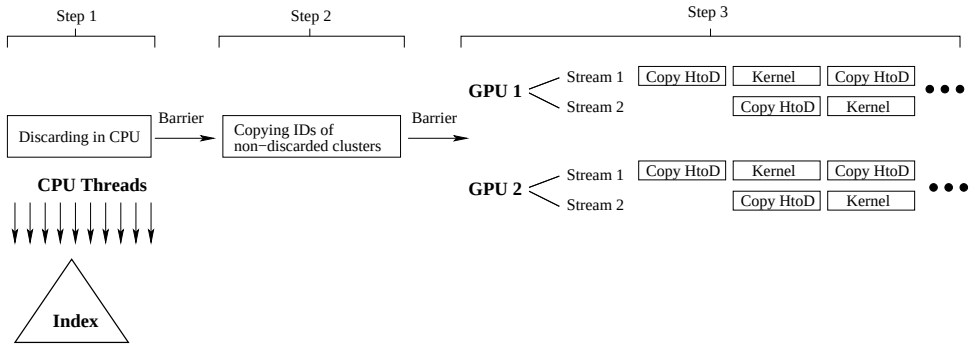


Figura A.23: Esquema de la estrategia multi-pipeline.

Debido a que los algoritmos de esta sección están diseñados para lidiar con grandes bases de datos, se ha utilizado la base de datos *CoPhIR* (Content-based Photo Image Retrieval) [6]. Esta consiste de metadatos extraídos de Flickr. Es una colección de 106 millones de imágenes, y por cada imagen cinco descriptores MPEG-7. Para el propósito de esta tesis sólo se utilizó por cada imagen el descriptor *Color Structure*, el que representa un vector de dimensión 64. Se usó la distancia euclidiana como función de distancia. Los radio utilizados son aquellos necesarios para recuperar el 0.01 %, 0.1 % y 1 % de elementos de la base de datos por consulta.

Utilizar un archivo de consultas real, obtenido de un motor de búsqueda por similitud en producción sería de gran utilidad para medir distintos parámetros, ya sea el comportamiento de las evaluaciones de distancia, localidad o regularidad en el acceso a memoria, entre otros. Según nuestro conocimiento, no existe ningún archivo de consultas real para búsqueda por similitud en imágenes. Pero, recientemente un sitio web público fue presentado en [46], en donde se aplica el motor de búsqueda MUFIN [59], que es usado por muchos usuarios alrededor del mundo. De este sitio web se obtuvo el archivo de consultas, que representan las consultas procesadas durante varios días. Se usaron 30,000 consultas que son representadas

por su descriptor *Color Structure* de dimensión 64. Hemos hecho público este archivo de consultas en [1].

Como en todos nuestros previos experimentos, para todas las estrategias los kernels son lanzados con un CUDA Block por consulta, y cada CUDA Block procesa una consulta diferente.

Las Figuras A.24(a), A.24(b) y A.24(c) muestran el tiempo de ejecución acumulado de los índices *LC* y *LSC* combinados con las estrategias de pipeline. Por ejemplo, en la primera barra de la Figura A.24(a), el tiempo de ejcución de la estrategia *1-Stage* procesando consultas en lotes de Q=154 es 11.7 segundos, con Q=98 es 16.2 segundos, y con Q=28 es 46.4 segundos. Se procesan las consultas en lotes de 28,98 y 154 porque son múltiplos de 14, que es el número de multiprocesadores en nuestra GPU, y tomando en cuenta que procesamos cada consulta con un CUDA Block distinto, un mútliplo de 14 mejora el balance de carga de CUDA Blocks entre los multiprocesadores.

Las primeras tres columnas de las figuras son versiones del *LC* combinados con los pipelines, y las últimas tres columnas son versiones del *LSC* combinado con los pipelines. La primera columna fue tomada como estrategia de referencia, y representa la estrategia *1-Stage* (Sección A.5.1.2). Esta utiliza el índice *LC*, y tras cargar N clusters a *device memory*, se lanza un kernel para procesarlos (N es el número de clusters que caben en *device memory*). La segunda columna (*1-Stage Pipe*) representa a la estrategia *1-Stage*, pero utilizando el pipeline copias-kernel descrito en la Sección A.5.2.3, por lo que tras cargar $N/2$ clusters en *device memory*, se lanza un kernel para procesarlos. La tercera columna (*1-Stage Pipe CPU-GPU*) es similar a la segunda, pero también implementa el pipeline CPU-GPU (Sección A.5.2.2), donde los threads que se ejecutan en núcleos de CPU intentan

descartar clusters en paralelo con el proceso en GPUs del lote de consultas previo. La cuarta columna (*LSC N-C*) representa al índice *LSC* (SecciónA.5.2.1), con *N* clusters por supercluster, y tras de cargar un supercluster a *device memory*, se lanza un kernel para procesarlo. La quinta columna (*LSC N/2-C Pipe*) representa al *LSC* con $N/2$ clusters por supercluster, y utilizando el pipeline copias-kernel, por lo que tras cargar un supercluster a *device memory* por un CUDA stream, se lanza un kernel para procesarlo en el mismo stream. La última columna (*LSC N/2-C Pipe CPU-GPU*) representa a la estrategia *Multi-pipeline* descrita en la Sección A.5.2.4, que utiliza el índice *LSC* con ambos pipelines.

Nuestra estrategia de referencia, etiquetada *1-Stage* en las figuras logran el peor rendimiento en todas las bases de datos para todos los *Q*. La estrategia *1-Stage Pipe* supera a la anterior, porque es capaz de explotar mejor los motores de copia y de ejecución de kernels de la GPU, usándolos todo el tiempo mediante el pipeline. Por lo tanto, se ocultan latencias en las transferencias de memoria y reducimos el tiempo de ejecución del algoritmo de búsqueda. La estrategia *1-Stage Pipe CPU-GPU* presenta mejor rendimiento que las dos anteriores, porque reduce la cantidad de clusters copiados a *device memory*, y también porque se ejecuta ese descarte mientras las GPUs procesan otro conjunto de clusters. La estrategia *LSC N-C*, a pesar de ejecutar más evaluaciones de distancia que el *LC* obtiene un mejor rendimiento, debido a su manejo más eficiente del ancho de banda de la GPU. Como se mencionó en la Sección A.5.2.1, se utiliza sólo una instrucción de copia para transferir un cluster o un supercluster en el caso del *LSC*, porque ellos son las unidades de transferencias del *LC* y *LSC* respectivamente. Pero, un supercluster es de mayor tamaño que un cluster, y el hecho de transferir cantidades mayores de datos en cada instrucción de copia otorga la ventaja al *LSC*. La estrategia *LSC*

N/2-C Pipe alcanza mejor rendimiento que las anteriores, porque la implementación del pipeline copias-kernel oculta latencias al usar CUDA streams. Finalmente, la estrategia etiquetada como *LSC N/2-C Pipe CPU-GPU*, que combina el *LSC* con ambos pipelines alcanza el mejor rendimiento. La ventaja de utilizar el pipeline CPU-GPU en el *LSC* es más evidente con Q=28, porque mientras mayor sea Q, mayor será el descarte de clusters. Esto último parece indicar un cierto grado de localidad cuando procesamos un pequeño lote de consultas, pero se pierde cuando el tamaño del lote crece. Sin embargo, dado que el número de transferencias es creciente debido a un reducido Q, los beneficios de esta localidad no son suficientes para mejorar el rendimiento global.

A.6. Conclusiones

La presente tesis ha propuesto un conjunto de algoritmos y estrategias para resolver búsqueda por similitud en espacios métricos, utilizando diferentes plataformas paralelas.

En la primera parte de la tesis, hemos utilizado una plataforma multi-core, donde hemos encontrado que es más conveniente el uso de ciertas estrategias dependiendo del tráfico de consultas entrantes. Se obtuvieron speed-up cercanos al óptimo con respecto a la versión secuencial. En la segunda parte, hemos utilizado una tarjeta gráfica NVIDIA GPU (Graphic Process Units), donde hemos propuesto y mapeado un conjunto de métodos de indexación y de búsqueda exhaustiva para resolver consultas por similitud, explotando eficientemente la jerarquía de memoria de la GPU. Hemos alcanzado un alto speed-up sobre la versión multi-core. En la tercera parte de esta tesis, hemos utilizado una plataforma multi-GPU, extendiendo

nuestros previos algoritmos en 1 GPU. Hemos cubierto los casos cuando la base de datos cabe completamente en memoria de la GPU, y tambiñen cuando la base de datos es suficientemente grande para no caber en la memoria de la GPU, que es un caso más real para bases de datos en producción.

A continuación presentamos las principales conclusiones para cada plataforma paralela.

Conclusiones sobre una plataforma multi-core

Hemos utilizado un conjunto de índices representativos y muy usados en la literatura técnica para mostrar lo genérico de nuestros algoritmos. Estos algoritmos implementan procesamiento multi-thread asíncrono (estrategia *Local*), y procesamiento bulk-sincrónico (estrategias *Bulk-Circular*, *Bulk-Local* y *Bulk-Critical*), siendo estas últimas una implementación del modelo BSP.

La estrategia *Bulk-Critical* mostró el peor rendimiento, debido principalmente al ato número de accesos a regiones críticas, y su alto costo asociado.

La estrategia *Local* mostró el mejor rendimiento bajo escenarios de alto tráfico de consultas. En esta estrategia cada thread procesa su consulta de forma independiente e incomunicada del resto de los threads.

La estrategia *Bulk-Circular* mostró el mayor rendimiento en escenarios de bajo tráfico de consultas. En esta estrategia cada thread distribuye el trabajo que implica la solución de una consulta entre los demás threads. Esto obtiene ventaja con un bajo tráfico de consultas porque reduce el tiempo de ociosidad de threads que esperan por una nueva consulta.

De acuerdo a los resultados previos, se propuso una estrategia *híbrida*, que es capaz de cambiar de modo de operación entre las estrategias *Local* y *Bulk-Circular*, dependiendo del tráfico de consultas actual del sistema. Se obtuvieron speed-ups cercanos a 8x con un servidor de 8 núcleos.

También, se compararon dos diferentes distribuciones de la base de datos. La primera, distribuye los elementos de la base de datos entre threads, y cada thread crea su propio índice. La segunda, mantiene sólo un índice global en memoria. Esta última mostró el mejor rendimiento, debido a la calidad de los centros (o pivotes) globales, mejorando la eficiencia del descarte de elementos.

Conclusiones sobre 1-GPU

Se propusieron y compararon diferentes algoritmos de búsqueda indexada y exhaustiva para los tipos de consultas por *rango* y *k*NN. Hemos utilizado los índices métricos *Lista de Clusters* (*LC*) y *SSS-Index* debido a: (1) sus buenos resultados en trabajos previos; (2) son capaces de almacenar sus elementos en matrices, y (3) presentan buena regularidad en el acceso a memoria. Los últimos dos puntos son características convenientes cuando se utiliza una GPU. En nuestra exploración encontramos que algunos parámetros óptimos en GPU son muy distintos a aquellos usados en computación secuencial; en particular el *SSS-Index* alcanza su mejor rendimiento con sólo 1 pivote para la base de datos de vectores.

Debido a la complejidad y restricciones de la GPU, encontramos diferentes problemas para ambos tipos de consultas, por *rango* y *k*NN, por lo que se aplicaron diferentes estrategias de paralelización para cada uno de ellos.

Con respecto a nuestras propuestas para resolver consultas *k*NN, se utiliza un conjunto de heaps almacenados en la memoria de la GPU para mantener los *K* elementos más cercanos a la consulta a través el proceso de búsqueda. Posteriormente, un warp y el primer thread del CUDA Block reduce los elementos de los heaps en sólo uno, almacenado en memoria compartida de la GPU, que posee los resultados finales.

Obtuvimos mejores resultados que métodos anteriores basados en ordenamiento. También, se muestra que sobre consultas *k*NN el método de *rango creciente* es más adecuado para ser usado en GPU que el método decreciente, que es la situación opuesta a lo que sucede en computación secuencial.

En ambos tipos de consultas, el índice *LC* alcanzó el mejor rendimiento, dada su buena regularidad y patrón de accesos a memoria de la GPU. Se alcanzó un speed-up de hasta 466x sobre el algoritmo de fuerza bruta secuencial.

Conclusiones sobre una plataforma multi-GPU

Se dividió esta parte en dos secciones con diferentes hipótesis iniciales: (1) el primer caso asume que toda la base de datos cabe completamente en la memoria de las GPUs, y (2) el segundo caso asume que sólo una porción de la base de datos cabe en memoria de la GPU.

El primer caso, cuando la base de datos cabe en memoria de la GPU, se propusieron y compararon diferentes estrategias para el índice *LC*, exponiendo las dificultades de lidiar con este tipo de entorno paralelo. Se obtuvo un speed-up superlineal sobre la versión de 1 GPU, el que se explica porque mientras mayor sea la cantidad de GPUs, mayor será el occupancy, lo que implica mayor número

de threads activos y mejor rendimiento en cada GPU. También, validamos nuestras propuestas en el contexto de sistemas de tiempo real, cuando no es aceptable esperar por miles de consultas para llenar el sistema antes de procesarlas en paralelo, concluyendo que las GPUs pueden ser utilizadas para procesamiento de consultas on-line en espacios métricos, como una alternativa de alto rendimiento y de bajo costo comparado con implementaciones multi-CPU tradicionales.

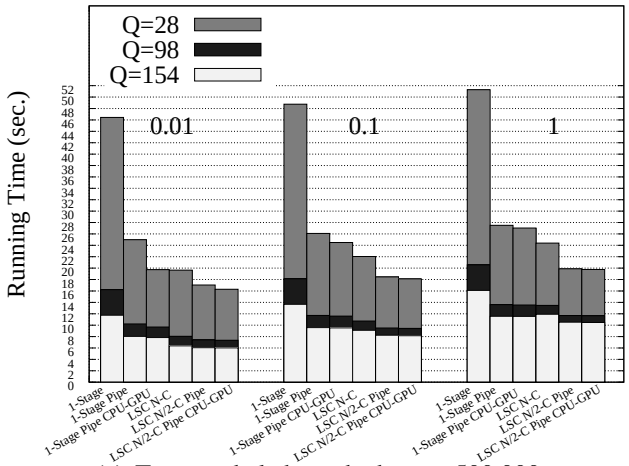
En la segunda sección, cuando la base de datos es suficientemente grande para no caber en memoria de la GPU, se propuso un índice jerárquico multi-nivel basado en el *LC*, llamado *Lista de Superclusters* (*LSC*). El *LSC*, compuesto de *superclusters*, ha sido diseñado para un buen rendimiento en GPUs. Un supercluster está formado por un centro, un radio cobertor, y sus elementos, pero con estos últimos se crea un índice *LC* dentro de cada supercluster. El agrupar clusters en superclusters permite un rápido descarte a nivel de CPU, y usando un supercluster como la unidad mínima de transferencia se asegura un uso eficiente del ancho de banda. Con el objetivo de lidiar con las transferencias de memoria, se implementó un pipeline híbrido entre CPU y GPU. Las CPUs ejecutan un primer paso de descarte para un lote de consultas Q_i, mientras en paralelo las GPUs están terminando de procesar el lote de consultas previo Q_{i-1}. Además, las transferencias CPU-GPU y las ejecuciones de kernels también se colocaron en un pipeline, usando *CUDA streams* y copias asíncronas. Esto último implica que la latencia por transferencias se oculte casi completamente; de hecho, aún si todo el índice es copiado por cada lote de consultas, la latencia total expuesta puede ser aún menor que cuando se transfiere toda la base de datos sólo una vez. El archivo de consultas utilizado en esta sección es también una contribución, porque según nuestro conocimiento no existe un archivo público de consultas reales para búsqueda por similitud en imágenes,

pero en esta sección se hace público el primero de ellos.

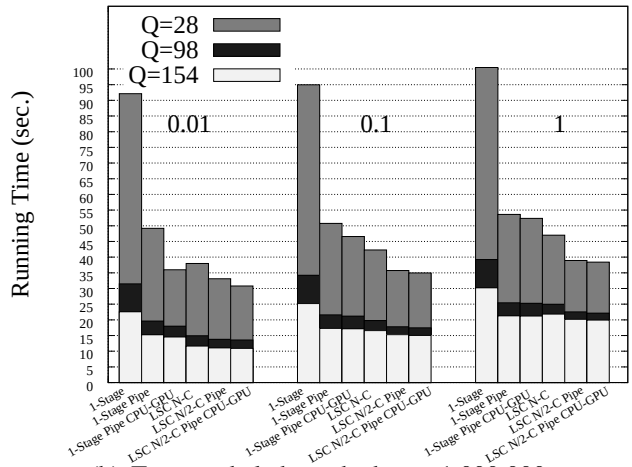
A.7. Trabajo Futuro

Una serie de propuestas quedan para el desarrollo futuro, entre ellas:

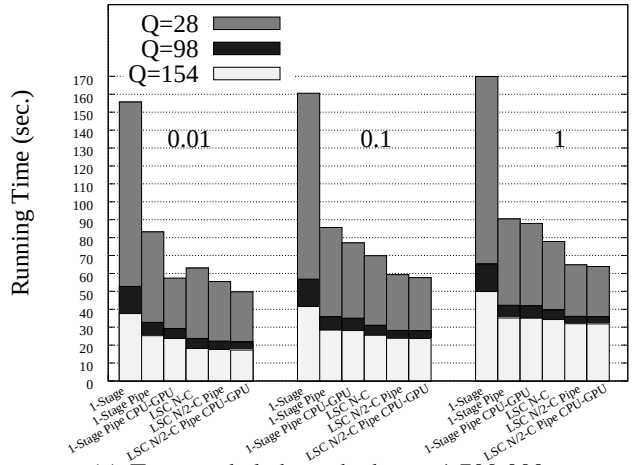
- Extender nuestros algoritmos para ser aplicados sobre una plataforma de memoria distribuida.
- Analizar y evaluar el rendimiento de estructuras métricas de tipo árbol, que podrían explotar eficientemente la jerarquía de memoria de la GPU.
- Diseñar algoritmos que exploten el resto de los recursos de la jerarquía de memoria en GPU.
- Analizar el uso de distintos métodos en GPU para realizar búsquedas eficientes en espacios de alta dimensión.
- Ampliar los algoritmos propuestos para implementar dinamismo en las estructuras de datos utilizadas en GPU.
- Evaluar el impacto de utilizar otras herramientas de programación diferentes de CUDA, como OpenCL, OpenACC, y otros basados en nuevos modelos de programación como MPI/OmpSs.
- Ampliar la integración de nuestros algoritmos en GPU para ser utilizados en otras plataformas heterogéneas, con el uso por ejemplo de FPGAs.



(a) Tamaño de la base de datos = 500,000



(b) Tamaño de la base de dato = 1,000,000



(c) Tamaño de la base de dato = 1,700,000

Figura A.24: Tiempo de Ejecución de los índices *LC* y *LSC* combinados con los pipelines copias-kernel y CPU-GPU.

Appendix B

List of Publications

Journal Papers

1. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto, M. Marin,
   "Range query processing on single and multi GPU environments",
   Computers and Electrical Engineering. ISI Journal. *In press*.

Conference Papers

1. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto, P. Zezula,
   "Multi-level clustering on metric spaces using a multi-GPU platform",
   In 19th International European Conference on Parallel and Distributed Computing (Euro-Par 2013). Springer, LNCS. Aachen, Germany, August 2013.

2. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto, M. Marin,
   "Range query processing in a multi-GPU environment",
   In 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012). IEEE. Madrid, Spain, July 2012.

3. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto, M. Marin,
   "kNN Query Processing in Metric Spaces using GPUs",
   In 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011). Springer, LNCS. Bordeaux, France, Sept. 2011.

4. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto,
   "Query Processing in Metric Spaces using GPUs",
   XII Jornadas de Paralelismo, Tenerife, Spain, Sept. 2011.

5. G.V. Costa, R. Barrientos, M. Marin and C. Bonacic,
   "Scheduling Metric-Space Queries Processing on Multi-Core Processors",

In 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010). IEEE CS. Pisa, Italy, Feb. 2010.

6. R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto,
"Heap-Based k-Nearest Neighbor Search on GPUs",
XXI Jornadas de Paralelismo, Valencia, Spain, Sept. 2010.

Bibliography

[1] *http://www.ribarrie.cl/Programs.html*.

[2] Google goggles. In *http://www.google.com/mobile/goggles/*.

[3] M-tree project: *http://www-db.deis.unibo.it/Mtree/*.

[4] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixedqueries trees. In *5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[5] Richard Bellman. *Adaptive control processes: a guided tour*. A Rand Corporation Research Study Series. Princeton University Press, 1961.

[6] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. Cophir: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627, 2009.

[7] Sergei Brin. Near neighbor search in large metric spaces. In *the 21st VLDB Conference*, pages 574–584. Morgan Kaufmann Publishers, 1995.

[8] Nieves R. Brisaboa, Antonio Fariña, Oscar Pedreira, and Nora Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *ISM*, pages 881–888, 2006.

[9] Nieves R. Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *SOFSEM*, pages 186–197, 2008.

[10] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science (ICCS)*, volume 3994, pages 196–199. Springer, 2006.

[11] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24, 2009.

[12] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2008.

[13] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[14] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 38–46, Los Alamitos, USA, 1999. IEEE CS Press.

[15] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[16] Edgar Chávez and Gonzalo Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[17] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José L. Marroquín. Searching in metric spaces. In *ACM Computing Surveys*, pages 33(3):273–321, September 2001.

[18] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[19] V. Gil Costa and M. Marin. Distributed sparse spatial selection indexes. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 440–444, Toulouse, France, February 2008. IEEE Computer Society.

[20] V. Gil Costa, M. Marin, and N. Reyes. An empirical evaluation of a distributed clustering-based index for metric space databases. In *24th International Conference on Data Engineering Workshops (ICDE 2008)*, pages 386–393, Cancún, México, April 2008. IEEE Computer Society.

[21] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[22] CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation.

[23] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Informations Systems*, 12(2):171–175, 1987.

[24] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. Speeding up spatial approximation search in metric spaces. *ACM Journal of Experimental Algorithmics (JEA)*, 14:article 3.6, 2009. 21 pages, doi: http://doi.acm.org/10.1145/1498698.1564506.

[25] K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[26] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. *Computer Vision and Pattern Recognition Workshop*, 0:1–6, 2008.

[27] Veronica Gil-Costa, Mauricio Marin, and Nora Reyes. Parallel query processing on distributed clustering indexes. *Journal of Discrete Algorithms*, 7(1):3–17, 2009.

[28] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[29] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[30] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. pages 151–155, Huangshan, China, 2009.

[31] Levi B. Larkey and Arthur B. Markman. Processes of similarity judgment. *Cognitive Science*, 29(6):1061–1076, 2005.

[32] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[33] M. Marin. Range queries on distributed spatial approximation trees. In *IASTED International Conference on Databases and Applications*, pages 140–144, Innsbruck, Austria, February 2005. IASTED/ACTA Press.

[34] M. Marin and V. Gil Costa. (sync—async)$^+$ mpi search engines. In *14th European PVM/MPI User's Group Conference (EuroPVM/MPI 2007)*, volume 4757 of *Lecture Notes in Computer Science*, pages 117–124, Paris, France, 2007. Springer.

[35] M. Marin, V. Gil Costa, and Carolina Bonacic. A search engine index for multimedia content. In *14th International European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, volume 5168 of *Lecture Notes*

in Computer Science, pages 866–875, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[36] M. Marin, R. Uribe, and R.J. Barrientos. Searching and updating metric space databases using the parallel egnat. In *7th International Conference on Computational Science (ICCS 2007)*, volume 4487 of *Lecture Notes in Computer Science*, pages 229–236, Beijing, China, May 2007. Springer.

[37] Mauricio Marin, Flavio Ferrarotti, and Verónica Gil-Costa. Distributing a metric-space search index onto processors. In *39th International Conference on Parallel Processing*, ICPP 2010, pages 433–442, San Diego, USA, September 2010. IEEE Computer Society.

[38] Mauricio Marin, Veronica Gil-Costa, Carolina Bonacic, Ricardo Baeza-Yates, and Isaac D. Scherson. Sync/async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 36(4):153 – 168, 2010.

[39] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[40] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[41] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734 – 747, 2011.

[42] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[43] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[44] H. Noltemeier. Voronoi trees and applications. In *International WorkShop on Discrete Algorithms and Complexity*, pages 69–74, Fukuoka, Japan, 1989.

[45] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex schemes of geometric object. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203, Springer-Verlag 1992.

[46] David Novak, Michal Batko, and Pavel Zezula. Generic similarity search engine demonstrated by an image retrieval application. In *32nd ACM SIGIR Conference on Research and Development in Information Retrieval*, page 840, Boston, MA, USA, 2009. ACM.

[47] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. Technical report, 2010.

[48] NVIDIA Corporation. *CUDA C Best Practices Guide*, 4.1 edition, january 2012.

[49] Roberto Uribe Paredes, Pedro Valero-Lara, Enrique Arias, José L. Sánchez, and Diego Cazorla. A gpu-based implementation for range queries on spaghettis data structure. In *Computational Science and Its Applications (ICCSA 2011)*, volume 6782 of *Lecture Notes in Computer Science*, pages 615–629, Santander, Spain, June 2011. Springer.

[50] P.J. Phillips, P.J. Flynn, T. Scruggs, K.W. Bowyer, Jin Chang, K. Hoffman, J. Marques, Jaesik Min, and W. Worek. Overview of the face recognition grand challenge. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 947–954 vol. 1, June 2005.

[51] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[52] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, California, USA, May 1995. ACM Press.

[53] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[54] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[55] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, pages 40:175–179, 1991.

[56] R. Uribe, G. Navarro, R.J. Barrientos, and M. Marin. An index data structure for searching in metric space databases. In *6th International Conference on Computational Science (ICCS 2006)*, volume 3991 of *Lecture Notes in Computer Science*, pages 611–617, Reading, UK, May 2006. Springer.

[57] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[58] E. Vidal. An algorithm for finding nearest neighbor in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[59] Pavel Zezula. Multi feature indexing network mufin for similarity search applications. In *38th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2012)*, volume 7147 of *LNCS*, pages 77–87. Springer, 2012.

[60] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

List of Figures

List of Tables